This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

Automated Generation and Evaluation of JMH Microbenchmark Suites from Unit Tests

Mostafa Jangali, Yiming Tang*, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, Weiyi Shang

Abstract—Performance is a crucial non-functional requirement of many software systems. Despite the widespread use of performance testing, developers still struggle to construct and evaluate the quality of performance tests. To address these two major challenges, we implement a framework, dubbed *ju2jmh*, to automatically generate performance microbenchmarks from JUnit tests and use mutation testing to study the quality of generated microbenchmarks. Specifically, we compare our *ju2jmh* generated benchmarks to manually-written JMH benchmarks and to automatically generated JMH benchmarks using AutoJMH framework, as well as directly measuring system performance with JUnit tests. For this purpose, we have conducted a study on three subjects (`Rxjava`, `Eclipse-collections`, and `Zipkin`) with ∼454K *source lines of code* (SLOC), 2,417 JMH benchmarks (including manually-written and generated AutoJMH benchmarks) and 35,084 JUnit tests. As a result, the *ju2jmh* generated JMH benchmarks consistently outperform using the execution time and throughput of JUnit tests as a proxy of performance and JMH benchmarks automatically generated using AutoJMH framework while being comparable to JMH benchmarks manually written by developers in terms of tests' stability and ability to detect performance bugs. Nevertheless, *ju2jmh* benchmarks are able to cover more of the software applications than manually-written JMH benchmarks during the microbenchmarking execution. Furthermore, *ju2jmh* benchmarks are generated automatically, while manually-written JMH benchmarks requires many hours of hard work and attention, therefore our study can reduce developers' effort to construct microbenchmarks. In addition, we identify three factors (too low test workload, unstable tests and limited mutant coverage) that affect a benchmark's ability to detect performance bugs. To the best of our knowledge, this is the first study aimed at assisting developers in fully automated microbenchmark creation and assessing microbenchmark quality for performance testing.

Index Terms—Performance, Performance Testing, Performance Microbenchmarking, JMH, Performance Mutation Testing

1 Introduction

PERFORMANCE is a crucial non-functional requirement of software systems. Performance can directly impact user-perceived software quality while using systems. Performance testing [1] aims to assess several non-functional attributes of systems, such as the execution duration of a process, response time, stability, and resource usage. However, in order to evaluate overall software systems, state-of-the-art performance testing practices typically involve relatively large-scale and long-running tests [2], which are time-consuming, difficult to be fully automated, and not easy to reconcile with continuous integration (CI) practices [3]. To address these issues, several approaches of performance unit testing, such as performance microbenchmarking, have been proposed for the precise performance evaluation of small-scale and isolated source code segments. Performance microbenchmarking is executed at a similar level of granularity as unit tests, e.g., method level or even a statement level [4], [5]. The goal of performance microbenchmarking frameworks is to detect performance bugs as early as possible by, for example, checking and testing each system build [6]–[8]. Due to the numerous challenges of performance microbenchmarking, such as unreliable results [9], [10], the need for in-depth knowledge of methodology [11], [12], and a lack of appropriate tooling [3], [13], [14], several performance microbenchmarking frameworks, including JMH in the Java ecosystem, have been proposed to automate performance microbenchmarking.

However, developers still encounter a number of difficulties when implementing performance microbenchmarking. On the one hand, developers struggle to construct correct microbenchmarks. Prior studies [14], [15] reveal that, as opposed to JUnit functional tests, only a few open-source projects use microbenchmarking frameworks to assess performance. The challenges of microbenchmark design [16] and configuration [17]–[19] (e.g., microbenchmarks may be "over-optimized" by JIT, resulting in muddled time measures.) hinder developers from effectively exploiting performance microbenchmarks [4], [16], [20], [21]. On the other hand, assuring microbenchmark quality is a significant concern. Microbenchmarks can be rendered ineffective and error-prone due to a variety of causes. For example, a prior study [5] highlights five problematic JMH practices which are common in open-source software.

In light of the fact that many projects lack microbenchmarks but do have unit tests [14], [15], we propose automatically deriving performance microbenchmarks from JUnit tests for performance testing, and speculate that such derived performance microbenchmarks are preferable to using JUnit tests directly to measure system performance (e.g., by measuring the execution duration of unit tests, as currently practiced in some related work [3], [15]). Specifically, we implement a framework dubbed *ju2jmh* that developers

* *Corresponding author.*
- *M. Jangali, Y. Tang, J. Yang and W. Shang are with Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.*
  E-mail: {m_jangal, t_yiming, jinqiuy, shang}@encs.concordia.ca
- *N. Alexandersson and P. Leitner are with Software Engineering Division, Chalmers | University of Gothenburg, Sweden.*
  E-mail: nicale@student.chalmers.se, philipp.leitner@chalmers.se

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

can use to automatically obtain a performance test suite, without manual development effort. To assuage developers' concerns about microbenchmark quality assurance (as mentioned above) and validate our hypothesis that derived microbenchmarks are superior to using JUnit tests to measure system performance directly, we then use mutation testing to examine our produced microbenchmarks and compare them to manually-written JMH benchmarks, automatically generated JMH benchmarks using AutoJMH framework, and directly measuring the execution time and throughput of JUnit tests. We design and implement a microbenchmark evaluation framework dubbed Performance Mutation Testing (PMT) framework, which includes five performance mutation operator APIs for reproducing performance bugs (i.e., inject performance bugs into the source code), based on five common performance bug patterns highlighted in recent studies [22], [23]. The framework enables us and developers to automatically reproduce performance bugs in any desired source code location and evaluate any specific performance microbenchmark.

We evaluate the quality of microbenchmark suites generated from *ju2jmh* by using PMT, and based on three projects with ∼454K *source lines of code* (SLOC), 2,417 JMH benchmarks (2,262 manually-written and 155 generated using AutoJMH) and 35,084 JUnit tests: `Rxjava`[1], `Eclipse-collections`[2], `Zipkin`[3].

Our study addresses the following research questions:

- **RQ1:** ***How stable are automatically generated performance microbenchmarks?***
  We use two metrics to assess microbenchmark stability, and both reveal that JMH benchmarks generated by *ju2jmh* (hereafter abbreviated as *ju2jmh* benchmarks) significantly outperform JUnit tests in all studied cases and are comparable to human-written JMH benchmarks and AutoJMH benchmarks. Because *ju2jmh* benchmarks are fundamentally JMH benchmarks, the conclusion that *ju2jmh* benchmarks are comparable to manually-written JMH benchmarks is reasonable. However, *ju2jmh* benchmarks are generated automatically, while JMH benchmarks are typically written by developers, therefore our tool can reduce developers' effort to construct microbenchmarks.
- **RQ2:** ***Can performance microbenchmarks detect artificial performance bugs from mutation testing?***
  Generated *ju2jmh* benchmarks outperform JUnit tests and AutoJMH benchmarks in artificial performance bug detection, but only in some cases outperform manually-written JMH benchmarks. However, *ju2jmh* benchmarks are able to cover (execute) more mutants (100%) than manually-written JMH benchmarks (66%) during the microbenchmark execution, suggesting that *ju2jmh* benchmarks can cover more of the applications. In general, *ju2jmh* benchmarks can detect a higher proportion of mutants exclusively than other tests.
- **RQ3:** ***Can performance microbenchmarks detect real performance bugs?***
  According to our experiment of a real-world performance bug and a similar artificially generated bug, *ju2jmh* performs better than JUnit in detecting the real bug and artificial bug. Furthermore, the artificial bug and real-world bug affect benchmarks in a very similar way.
- **RQ4:** ***What are the major factors affecting a microbenchmark's ability to detect performance bugs?***
  We identify three common factors that jeopardize a benchmark's ability to identify performance bugs: (1) Too low workload (this applies to generated *ju2jmh* and JUnit tests, but not to manually-written JMH benchmarks). (2) Unstable benchmarks. (3) Limited mutant coverage as a result of inappropriate execution times (neither too low nor too high).

In conclusion, we compare our generated *ju2jmh* benchmarks with manually-written JMH benchmarks and with automatically generated JMH benchmarks using AutoJMH framework, as well as using the execution time and throughput of JUnit tests as a proxy of performance. The *ju2jmh* benchmarks consistently outperform JUnit tests while being comparable to manually-written JMH benchmarks and AutoJMH benchmarks in terms of microbenchmarks' stability and ability to detect performance bugs. However, the generated *ju2jmh* benchmarks can cover more of the applications than manually-written JMH benchmarks. In addition, we automate our approach for creating and assessing *ju2jmh* benchmarks, whereas human-written JMH benchmarks require a considerable amount of human effort. Moreover, although *ju2jmh* benchmarks and AutoJMH benchmarks are both generated automatically, AutoJMH has several serious restrictions. As a result, our technique successfully aids developers in the microbenchmark construction, and microbenchmark quality is assured. To further assess microbenchmark quality, we also reveal three factors that affect microbenchmarks' ability to detect bugs for future microbenchmark enhancement.

Paper organization

Section 2 introduces background on performance microbenchmarking, while Section 3 discusses the related studies on generating, assessing, and improving performance microbenchmarks. In Section 4, we present the overview of our study, which is comprised of two phases discussed in Section 5 and Section 6, respectively. Section 7 presents the experiments for our study. In Section 8, we clarify threats to the validity of our study. We conclude our study and discuss future work in Section 9.

2 Background

This section provides an overview of the performance microbenchmarking in the Java ecosystem and terminology used throughout this paper.

2.1 Performance Microbenchmarking in Java.

Performance testing [1] aims to assess several non-functional attributes of systems, such as the execution duration of a process, response time, stability, and resource

[1]https://github.com/ReactiveX/RxJava
[2]https://github.com/eclipse/eclipse-collections
[3]https://github.com/openzipkin/zipkin

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

usage. In this study, we focus on unit-level (e.g., method) performance testing of systems, known as performance microbenchmarking. In contrast to system-level testing, which is for assessing the overall performance, and load (or stress) testing, which is an end-to-end testing approach, performance microbenchmarking aims to evaluate small units of systems, like the throughput of a method, or the performance of an algorithm.

Over the past few years, many Java frameworks such as Caliper [24], JMH [25], AutoJMH [16], or JUnitPerf [26] have provided microbenchmarking tooling. However, evidence [14] in practice confirms that performance microbenchmarking is not yet as established as functional unit testing or system performance testing. Users of these frameworks must either create their own performance microbenchmarks from scratch (e.g., JMH benchmarks), automatically generate and run JMH benchmarks by using an existing tool (e.g., AutoJMH benchmarks), which works only under certain situations, or scaffold existing microbenchmarks (e.g., JUnitperf that applied on JUnit test cases). Only a few open-source projects deploy any microbenchmarking frameworks, and the projects often use JUnit functional tests repeatedly to evaluate performance [14], [15]. Specifically, they execute JUnit tests for a specific number of times or a duration for an effective measurement of elapsing time or throughput.

2.2 Java Microbenchmarking Harness (JMH)

Due to the numerous challenges of performance microbenchmarking, such as unreliable results [9], [10], the need for in-depth knowledge of methodology [11], [12], and a lack of appropriate tooling [3], [13], [14], several performance microbenchmarking frameworks, including JMH in the Java ecosystem, have been proposed to automate performance microbenchmarking. JMH is a framework developed under the OpenJDK umbrella that enables users to design and run repeatable performance microbenchmarks. JMH, similar to JUnit, uses Java annotations to prepare the testing environment for evaluating specific *payloads* (i.e., a self-contained program that wraps the code segment that is essential to system performance [16]) through several metrics, such as the throughput of a method. A JMH benchmark class can contain several benchmarks, each of which is tagged by an annotation `@Benchmark` and is usually designed to test the methods of related source code classes.

Figure 1 depicts an example of a JMH benchmark class from `Eclipse-collections` project. The methods with annotations `@Setup` and `@TearDown` are used to prepare and clean up the microbenchmarking environment respectively (equivalent to JUnit test fixtures), while the method with `@Benchmark` (that is equivalent to `@Test` in JUnit) is used to evaluate the performance of a functionality of `MaxByIntTest` class. Benchmark fixtures can be optionally executed in either the class scope or the benchmark scope to prepare or clean up microbenchmarking environment. Methods tagged by `@Setup` , such as the method `setUp()` in the Figure 1, are executed before running the benchmarks (tagged by `@Benchmark`), and similarly, methods tagged by `@TearDown`, like the method `tearDown()` in the figure, are executed after running the benchmarks.

Test suite

```
public class MaxByIntTest extends AbstractJMHTestRunner
{
```

Test fixture

```
    private final Positions positions
                                = new Positions(SIZE).shuffle();
    private ExecutorService executorService;
    @Setup
    public void setUp()
    {
        this.executorService = Executors.
                                newFixedThreadPool(
                                    Runtime.getRuntime()
                                        .availableProcessors());
    }

    @TearDown
    public void tearDown() throws InterruptedException
    {
        this.executorService.shutdownNow();
        this.executorService.
                        awaitTermination(1L, TimeUnit.SECONDS);
    }
```

Test case

```
    @Benchmark
    public Position maxByQuantity_serial_lazy_direct_methodref_jdk()
    {
        return this.positions
                .getJdkPositions().stream()
                    .max(QUANTITY_COMPARATOR_METHODREF).get();
    }
}
```

Figure 1: A JMH benchmark example from `Eclipse-collections` project

JMH benchmarks can be designed to measure performance, such as throughput or execution time, with flexible and different configurations. However, a single JMH benchmark iteration can encompass repetitive workload executions. As a result, to facilitate the evaluation of microbenchmark quality, we use workload throughput over a particular time period (designated "measurement time" and set to 1 second by default, i.e., `@Measurement(time =1)`) as a metric for further study, rather than focusing on the details of repetitive control flow in the benchmarks. Accordingly, the execution time of benchmarks is calculated by $average\_execution\_time = \frac{1}{throughput}$. In our experiments, specifically, the metric is the number of operators executed per second for a microbenchmark.

In general, JMH benchmarks have more advantages than JUnit tests for evaluating performance. As previously mentioned, one common performance testing practice is to run JUnit in a loop to measure performance [14], [15]. Although JMH benchmarks also use a loop to execute the payload repeatedly, the elegant features provided in JMH benchmarks lead to more accurate performance measurement results. For example, JMH benchmarks perform warm-up iterations to stabilize the system in order to avoid data noise caused by the execution environment, can have benchmark fixtures to prepare and clean-up the testing environment, and easily handle running benchmarks through multiple iterations, threads, and forks.

2.3 Mutation Testing

Mutation testing is a technique that aims to assess and improve the efficacy of test suites in discovering faults [27]. This technique reproduces artificial bugs through several predefined rules (i.e., mutation operators) that help in measuring the efficiency of a test suite in detecting them [22]. Mutation testing for performance purposes, so-called performing mutation testing, is a novel practice that raises

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

many open challenges and questions to address. Recent studies [3], [22] deploy mutation testing to address the efficiency of performance benchmarks in discovering performance bugs.

2.4 Detecting Real-world Performance Bugs

A performance bug could be detected by the covering performance tests by searching for degradation in their test results. For example, if we measure throughput of the covering performance test and the throughput is decreased significantly enough (e.g., 1%, 5%, and 10%), we consider that the test detects some performance bugs. For mutation testing, the test is run on both the system version with and without artificial performance bugs, and the results from the two versions are compared to determine whether performance degradation exist or not (i.e., the mutant is killed or not). For real-world performance bug detection, the test is run on the parent system that is assumed to contain a bug and the child system with new source code changes where the bug fixes are applied. After comparing two versions of test results, we can detect the existence of the performance degradation as well (i.e., the real-world performance bug is presented in the parent system).

3 Related Work

In this section, we provide an overview of the related work through three different perspectives.

Performance microbenchmarking. Recent studies deem that performance testing frameworks should concentrate on integrating with standard CI tooling and facilitating effective testing construction techniques [3], [15], [28]. Performance microbenchmarking is a software solution for short-term performance testing by integrating CI techniques, however, it is less established and commonly used than unit testing or system performance testing. Only a few open-source projects use performance unit testing frameworks, whereas many others rely on JUnit functional tests to assess performance [14], [15]. Design [16] and configuration [17]–[19] challenges (e.g., microbenchmarks may be "over-optimized" by JIT, resulting in muddled time measures.) hinder developers from effectively exploiting performance microbenchmarks [4], [16], [20], [21]. To reduce difficulties, Ding *et al.* [8] utilize JUnit tests to assess the performance properties of systems. Rodriguez-Cancio *et al.* [16] propose *AutoJMH*, an automated approach that can take one source code segment and one covering unit test as input and generates a full JMH microbenchmark for that segment [16]. However, AutoJMH is limited to manually configuring specific single statements and has preconditions that prevent developers from testing a large portion of the source code. In addition, the tool is obsolete because it has not been maintained or updated since August 12, 2016. In this study, we implement an automated tooling framework, *ju2jmh*, for widely exploiting JMH performance microbenchmarks.

Empirical studies on performance bugs. Performance-related bugs are well-known as a threat to users' positive perception of software systems. Moreover, since performance bugs are usually difficult to reproduce, it takes a long time to detect and fix them, such as 1,075 days on average to discover and fix 36 performance bugs in Jin *et al.* [29]. Nistor *et al.* [13] disclose that, compared to reasoning about functional bugs, developers have little support for reasoning about performance bugs. In addition, more experienced developers are required to address performance bugs [30].

Recent studies have identified commonly reported performance bugs, which can serve as a guideline for reproducing and fixing bugs. Radu and Nadi [23] provide a dataset, *NFBugs*, including eight performance bug patterns from a total of 138 non-functional bug fixes in 67 open-source projects in Java or Python. Delgado-Pérez *et al.* [22] review prior empirical studies on performance bugs [31]–[36] and summarize seven performance anti-patterns. Laaber and Leitner [3] insert slowdowns (i.e., `Thread.sleep(...)`) into the most used API(s) of subjects to pause execution of the program and to simulate the presence of a performance bug.

Assessing the quality of performance tests. Leitner and Bezemer [15] and Stefan *et al.* [14] study Java open-source systems in terms of the quality and quantity of systems' JMH benchmarks. Recent studies [3], [37], [38] claim stability is a crucial quality attribute of performance tests. In addition, prior work [3], [22] deems the bug detection ability of performance tests as another essential quality factor. Ding *et al.* [8] discuss the ability of readily available unit tests in detecting reported performance issues. As a result, they reveal eight factors to make a (unit) test effective at discovering performance bugs in a release pipeline.

Recent studies [3], [22] deploy mutation testing to address the ability of JMH microbenchmarks in discovering reproduced performance bugs. Laaber and Leitner [3] propose an approach to assess JMH microbenchmarks in discovering injected slowdowns. Delgado-Pérez *et al.* [22] use seven performance anti-patterns as means to assess JMH microbenchmarks in discovering performance bugs. However, both studies have limited flexibility with several manual actions that hinder followers from effectively performing their approaches in practice. In this study, we propose the first performance mutation framework to assess the quality and efficiency of performance tests automatically.

4 Study Overview

This section provides an overview of our study methodology[4], which primarily consists of two phases: (1) generating microbenchmarks using *ju2jmh*[5]. (2) evaluating microbenchmarks using mutation testing. In the next two sections, we discuss each phase in detail.

Figure 2 depicts an overall workflow of our methodology. To prepare our experiment, we begin by searching through test packages to extract desired JMH microbenchmarks and JUnit test suites, as well as setting up the performance (unit) testing environment. We then deploy *ju2jmh* to generate JMH microbenchmarks from JUnit test suites. In addition, we also deploy AutoJMH framework to generate JMH benchmarks for performance testing evaluation. Given the produced microbenchmarks, we use our

[4]The data and scripts of our study are shared at https://github.com/senseconcordia/JU2JMH-PMT-TSE-2021.

[5]The tool is publicly available at: https://github.com/alniniclas/junit-to-jmh/

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

PMT framework to inject performance bugs into the source code, and evaluate the microbenchmarks' quality in terms of result variability and bug-detection ability. To further study the microbenchmarks' ability to detect performance bugs, we also investigate the factors that may affect the microbenchmarks' detection of them.

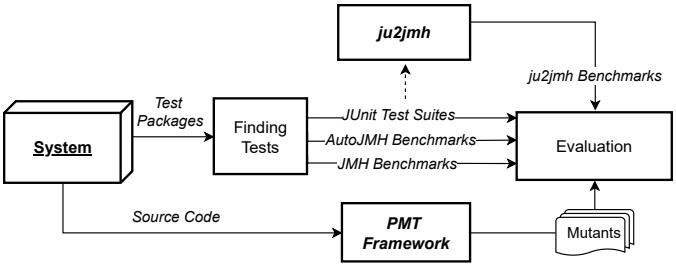


Figure 2: Our methodology to address the quality of performance microbenchmarks

Prior to beginning an experiment, preparation is needed. It necessitates the extraction of JMH microbenchmarks and JUnit test suites from the studied subjects for subsequent analysis.

Human-written JMH microbenchmarks. *JMH* is a Java microbenchmarking framework that has shown promise in properly scaffolding payloads [16]. Existing JMH microbenchmarks of systems are usually developed under the careful effort of the same developers who designed the system. Since designing JMH benchmarks necessitates deep knowledge of both system and the framework, developers are hesitant to exploit and maintain them. In this study, we have investigated existing human-written JMH microbenchmarks of systems. After building the systems, an executable `jar` is created by automated building tools to assist testers in executing JMH benchmarks of the system.

JUnit test suites. *JUnit* tests are designed to assess whether unit elements of the software system are functionally correct. Generally, JUnit tests do not tend to help measure the performance, as they are highly vulnerable to noisy environments. Ding *et al.* [8] utilize JUnit tests to evaluate the performance of tests. Therefore, to assess the performance of JUnit tests in our study, we measure test throughput by repeatedly executing unit tests for a given duration (e.g., one second). To do so, we deploy JUnit `@Rule` to monitor individual test execution and quantify the elapsed time in *nanoseconds*, allowing us to determine the throughput of a test method (i.e., `@Test`). In our experiments, we use a nested loop to measure the throughput of a JUnit test method 30 times to serve as a baseline for comparing different tests. The inner loop is used to measure the throughput of the test by executing the test as many times as the specific duration is reached. The outer loop is used to measure the throughput of the test multiple times, providing multiple data-points for the test.

AutoJMH microbenchmarks. Rodriguez-Cancio *et al.* [16] propose *AutoJMH*, an automated approach that can take one source code segment and one unit test that covers the segment as input and generates a full JMH microbenchmark for it [16]. Only single statements can be supplied as the source code segment input for AutoJMH and the input statements should be executed by at least one unit test. The JMH benchmarks generated by AutoJMH, also known as AutoJMH benchmarks, require a collection of variables as input and the variable values are extracted from the execution of covering unit tests from which the benchmarks were derived. The purpose of AutoJMH benchmarks is to measure the performance of input statements without running the entire program. In addition, the tool is obsolete because it has not been maintained or updated since August 12, 2016.

Regarding the similarities of AutoJMH and *ju2jmh*, they both use JUnit tests as input and then automatically generate JMH benchmarks. However, AutoJMH and *ju2jmh* have major differences. As defined above, AutoJMH executes JUnit tests for one time to gather required data for building JMH benchmarks, while *ju2jmh* benchmarks are designed to repeatedly execute JUnit tests, as payload, to measure their performance. Furthermore, AutoJMH benchmarks execute input statements without executing the entire program, while *ju2jmh* benchmarks execute input statements (i.e., JUnit tests) by actually executing the program. Lastly, AutoJMH requires manually specifying single statements as input for JMH benchmark building, which means that the number of generated JMH benchmarks is limited by the number of input statements. However, *ju2jmh* benchmarks are built on existing JUnit tests of programs, and the number of JUnit tests of a well-known program is usually quite large. As a result, in our experiment, the number of JMH benchmarks generated by AutoJMH is significantly lower than *ju2jmh*, which results in a much lower coverage of the program for AutoJMH.

5 Generating Microbenchmarks Using *ju2jmh*

To demonstrate the feasibility of generating JMH benchmarks from unit tests, we implement a tool dubbed *ju2jmh* that can automatically generate ready-to-execute JMH benchmarks from JUnit4 test suites. The main goal of *ju2jmh*'s design is to be able to generate functionally correct benchmarks from a wide variety of real-world unit tests while minimizing performance overhead and complexity.

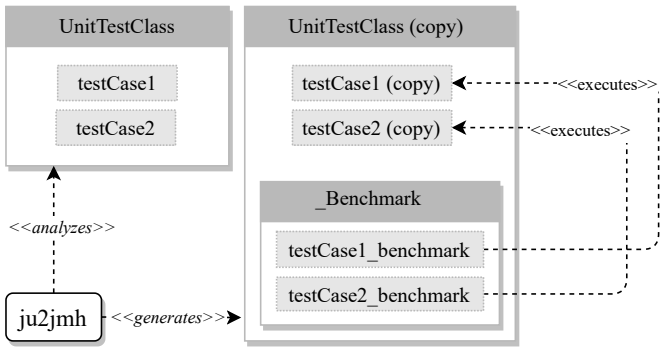


Figure 3: Overview of the *ju2jmh* approach and its inputs and outputs

Figure 3 depicts the basic benchmark generation procedure of *ju2jmh*. The procedure consists of two main steps: (1) **Analysis step**: the *ju2jmh* tool analyzes the existing unit test classes in order to identify individual JUnit test

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

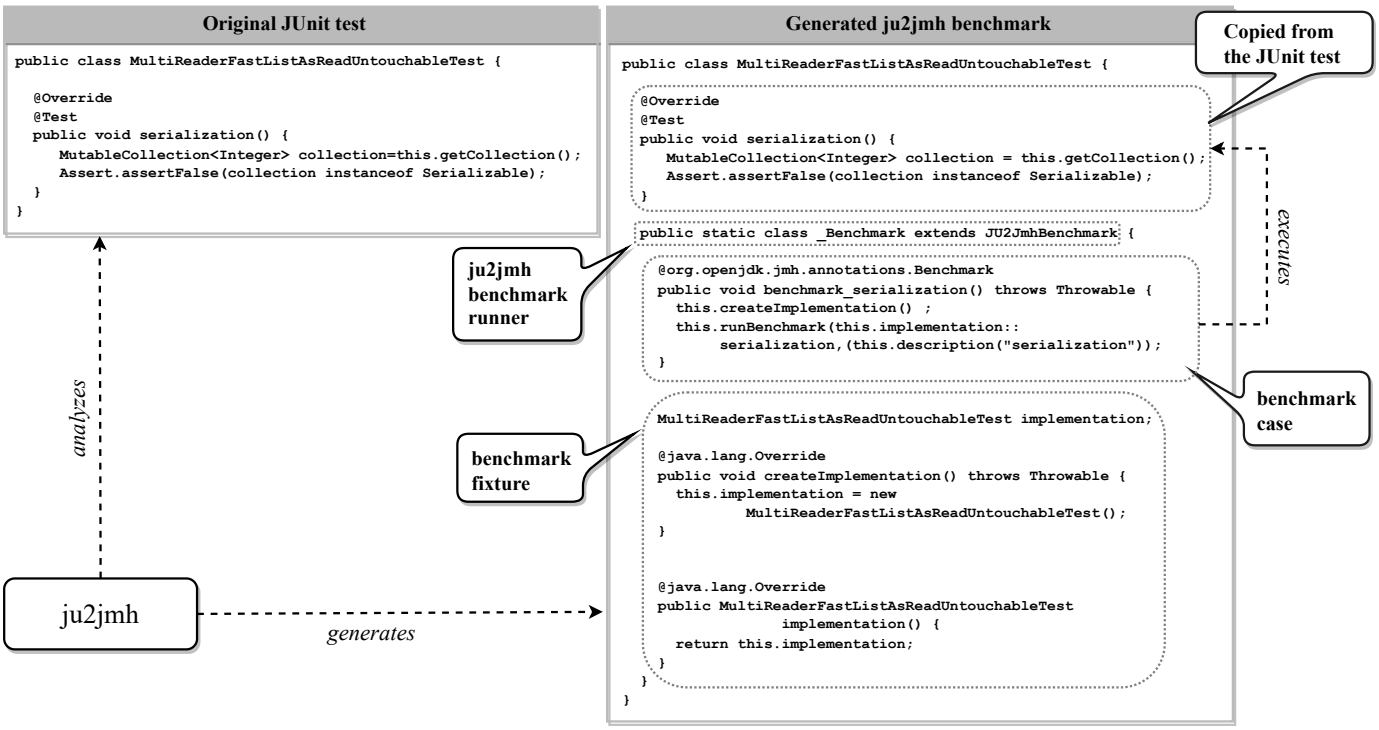


Figure 4: A real-life generated *ju2jmh* benchmark from `Eclipse-collections`, `MultiReaderFastListAsReadUntouchableTest.serialization()`, and how it actually performs JMH rules

methods and other relevant test features; (2) **Benchmark generation step**: the tool generates JMH benchmarks with each benchmark method responsible for repeatedly executing (a copy of) a single unit test method as its payload. For example, as shown in Figure 3, if *ju2jmh* is applied to a unit test class called `UnitTestClass`, which contains two unit test methods named `testCase1` and `testCase2`, *ju2jmh* first generates a copy of the `UnitTestClass` class, and then generates a JMH benchmark class called `_Benchmark` that is placed within this copy. The new benchmark class contains two JMH benchmark methods, `testCase1_benchmark` and `testCase2_benchmark`, whose responsibilities are to repeatedly execute `testCase1` and `testCase2` as their payloads.

In the analysis step, *ju2jmh* first identifies the individual JUnit test methods that are expected to be converted to JMH benchmarks. Next, *ju2jmh* detects any other features of the test methods that are required for proper execution, such as fixture methods, test rules, and expected exceptions, all of which should be handled explicitly by the generated benchmarks. The analysis is performed using *Apache Commons BCEL*[6] to statically inspect the bytecode of all relevant JUnit test classes (specified as input for the tool) in order to find test methods (annotated by `org.junit.Test`), fixture methods (annotated by `org.junit.Before`, `org.junit.After`, `org.junit.BeforeClass`, or `org.junit.AfterClass`), test rules (annotated by `org.junit.Rule` or `org.junit.ClassRule`), expected exceptions (extracted from the arguments of the `org.junit.Test` annotation), and ancestor classes containing any of the above.

In the benchmark generation step, *ju2jmh* uses the *JavaParser*[7] library to parse Java source code and generate the Abstract Syntax Tree (AST). This step consists of the following activities: (1) copying the AST of each relevant JUnit test class, (2) generating the AST for a corresponding `_Benchmark` class that is customized based on the information gathered in the analysis step, (3) inserting the generated benchmark class into the AST of the copied JUnit test class as a static member class, and (4) printing the results via a Java source file. The generated benchmark class contains a JMH benchmark method for each of the JUnit test methods in the JUnit class.

Each generated benchmark inherits the same superclass, which implements the functionality required to execute JUnit tests properly through the benchmark. The *ju2jmh* benchmark superclass contains methods to help instantiate and access a JUnit test class instance, methods to invoke fixture methods of JUnit test class (including all fixture methods from the superclasses), methods to apply all rules of the JUnit test class and its superclasses, and a benchmark execution method to execute the JUnit test methods. In the cases with the expected exceptions, the *ju2jmh* benchmark superclass executes a different benchmark execution method with the expected exceptions as an additional parameter.

In terms of the way *ju2jmh* works, class copies are preferred over direct accessing the classes due to subtle considerations. First, in the case where the tool creates a copy of the JUnit test class rather than directly referencing the original class, this processing allows future versions

[6]A Java bytecode analysis tool from The Apache Software Foundation: https://commons.apache.org/proper/commons-bcel/

[7]A tool for analyzing, transforming and generating Java codebase: https://javaparser.org/

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

of *ju2jmh* to make changes to the JUnit test methods in order to improve the generated benchmarks, for example by preventing optimizations such as dead code elimination. Moreover, rather than making the generated benchmark class a separate top-level class, it is inserted as a static member of the copy to reduce the risk of naming conflicts and to keep the benchmarks organized closely with the tests they use as payloads.

Our approach *ju2jmh* has been implemented in Java and integrated into the Gradle build system. *JavaParser* and *Apache Commons BCEL* are the primary libraries used for the unit test analysis and benchmark generation. The generated benchmarks are dependent on JMH [25] and JUnit4[8] libraries.

6 Evaluating Microbenchmarks Using Mutation Testing

In this section, we present the workflow of our PMT framework for evaluating unit-level performance tests (microbenchmarks). First, the framework implements and employs five performance mutation operators that are derived from common real-world performance bugs [3], [12], [22]. Second, it automatically injects performance bugs (based on the five mutation operators) into the analyzed software system and generates performance mutants (i.e., artificial performance bugs). Then, the generated performance mutants are executed against the to-be-evaluated performance microbenchmarks. Furthermore, to better classify between an actual slowdown (i.e., a performance bug) and a performance fluctuation, we leverage hierarchical re-sampling [9], [17] and two statistical measures, namely, *Relative Standard Deviation* (RSD) [17], [37], [38] and *Relative Confidence Interval of means* (RCI) [9], [17]. Last, the PMT framework computes the *mutation score*, which is defined as the proportion of the killed (detected) performance mutants, to assess the effectiveness of performance microbenchmarks in detecting performance bugs.

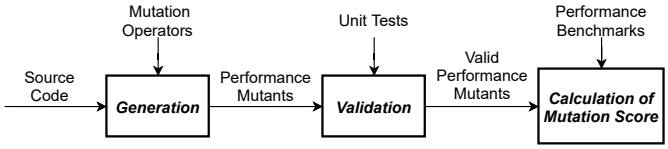


Figure 5: Overview of our Performance Mutation Testing framework.

6.1 Designing Mutation Operators for PMT

Our study aims at assisting developers in alleviating the challenges of constructing and evaluating performance tests rather than detecting performance bugs. To achieve our study goal, we use PMT to inject artificial performance bugs for further analysis. We evaluate performance tests by comparing the test results on the systems with and without performance bugs. Performance tests can help developers reveal the effects that mutants have on the systems, detect performance bugs and complement the omission of static tools that may treat some code as syntactically correct but cannot reveal its negative impact on performance.

A PMT framework requires specialized mutation operators for generating performance mutants (i.e., software versions with artificial performance bugs or slowdowns). In this study, we expose five performance mutation operators based on previous research on summarizing performance anti-patterns [3], [22], [23], which we then implement in our PMT framework.

In particular, Delgado-Pérez *et al.* [22] review prior empirical studies on performance bugs [31]–[36] and summarize seven performance anti-patterns, of which we employ two patterns for designing our performance mutation operators and exclude five others (three memory-related bug patterns and two for future work). Radu and Nadi [23] present the NFBugs dataset, which contains eight performance bug patterns discovered after analyzing 36 performance bug fixes from a total of 138 non-functional bug fixes in 67 open-source Java or Python projects. We adopt three performance anti-patterns from NFBugs and leave the remaining five as future work. Furthermore, Laaber and Leitner [3] propose inserting `Thread.sleep(...)` into source code to slow down program execution for performance mutation testing, which we also accept for the design of our performance mutation operators. We selected the five performance bug patterns because of the high availability of source code statements that a pattern can be applied to. Moreover, the five patterns do not require manipulating multiple lines of source code, and generated bugs scarcely tend to endanger the overall functionality.

In summary, we introduce a total of five performance mutation operators based on prior studies on analyzing performance bugs and implement them in our PMT framework. Such performance mutation operators are not specific to certain software systems and can be applied to any Java system without manual effort. Moreover, the PMT framework is extensible, allowing for the easy integration of new performance mutation operators.

Below we describe the five performance mutation operators in detail, with a summary in Table 1.

- **Primitive to Wrapper** (PTW). Replacing a primitive type (e.g., `long`) with its corresponding wrapper class (i.e., `Long`) can lead to performance bug [23], since primitive types are stored on the stack and provides faster access. We include the built-in wrapper classes for all primitive types, i.e., `Byte`, `Boolean`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Character`.
- **StringBuilder to StringBuffer** (STS). Replacing a `java.lang.StringBuilder` object with a `java.lang.StringBuffer` object can result in performance bug because `StringBuilder` is not synchronized [23]. Apart from the difference in performance, functionalities of these two classes are equivalent. Therefore, we perform the operator only on method-level to prevent propagation of changes.
- ***Enhanced For* Loops** (EFL). Replacing a traditional `for`-loop with a `for-each` loop to iterate over an array or a `Collections` class can introduce performance bug because of extra calls. The semantics of

[8] A framework to write repeatable Java unit tests: https://junit.org/junit4/

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

the code is usually unaffected by such replacements. However, if the loop counter is used in the `for`-loop, the replacement may cause performance bug due to the additional cost of calculating it.

- **Swap of Operands in Condition** (SOC). Reordering the two operands in a compound OR condition if the left operand is a variable or a Boolean literal (i.e., `True` and `False`), and the right operand is a method invocation. Forcing the evaluation of the right operand (i.e., a method invocation) by placing it as the left operand in a compound OR condition may introduce slowdown since the method invocation may not be executed originally as the right operand [22].
- **Simulation of Heavy-Weight Operations** (HWO). Injecting a delay (e.g, `Thread.sleep(t)`) can slow down the program execution. In our study, we consider injecting a delay as the first statement of each JMH microbenchmark (annotated by `@benchmark`). We chose the first statement as the candidate injection location since we expect to see that the delay is executed definitively, i.e., the injected delay cannot be affected by the data-flow and control-flow for the code in this method. For example, if a delay is injected into a branching statement, the delay may never be executed depending on the branching conditions. The HWO has been used in prior studies [3], [22] to simulate slowdowns, however, it does not represent real-world performance bugs. Nevertheless, we believe the HWO complements the other mutation operators based on real-world performance bugs and thus include HWO in this study.

6.2 Generating and Validating Performance Mutants

The PMT framework parses the source code of a target system into Abstract Syntax Trees (ASTs) and looks for opportunities to apply the five performance mutation operators on the parsed ASTs. For each applicable AST location and each mutation operator, the PMT framework generates one performance mutant. For example, the EFL mutation operator can be applied to a subset of traditional `for`-loop in one system, in `Rxjava`, the PMT framework finds 682 applicable locations and generates a total of 62 performance mutants.

Note that if a mutant is killed by any *functional test*, it is an invalid performance mutant since it breaks functionality. The system's regular behavior could be disrupted by the presence of such mutants, which leads to the system's performance not being properly measured. Therefore, in this study, we only consider performance mutants that do not change system functionality and require performance tests to detect. As a result, we introduce a validation step in the PMT framework: All the generated performance mutants are executed against the functional unit tests and are excluded from the further step (i.e., calculating mutation score) if one cannot make all the functional tests pass.

6.3 Calculating Mutation Score

Relying on the purpose of mutation testing in performance that is introducing performance bugs deliberately, we looked over the benchmark's results against a valid mutant to find significant differences. Recent studies [3], [22] have been advocating *statistical hypothesis tests* (e.g., Wilcoxon rank-sum test) to label a significant difference in results as a performance bug, slow down, or the presence of artificial bugs. However, prior studies [37] concluded that testing with Wilcoxon rank-sum tests is not a suitable vehicle for detecting performance degradation in cloud environments due to high false-positive reported.

To better classify between an actual performance slowdown and a performance fluctuation, we leverage hierarchical re-sampling [9], [17] and a statistical measure, i.e., **Relative Confidence Interval of means** (RCI) [9], [17].

For a benchmark that covers and executes any of generated performance bug, the PMT framework deploys ***pa*** [39] to estimate the throughput's RCI (of before and after the bug injection) with bootstrap (i.e., re-samples) [40], [41] using 10,000 bootstrap iterations [42]. Respectively, the size of a performance bug can be defined as $[1 - Upper_{RCI}]$ (in %), which reflects the effectiveness of mutant against the benchmark. If the bug size is significant enough (e.g., $\geq 5\%$), we can consider that the benchmark could detect the injected performance bug (i.e., a killed mutant).

Last, the PMT framework calculates the mutation score of performance microbenchmarks as the percentage of the killed performance mutants. The calculated mutation score helps evaluate the quality of one or a group of microbenchmarks. The microbenchmarks' quality (and efficiency) in discovering bugs increases as their mutation score rises.

One of the advantages of mutation testing is to find deficiencies in tests and improve them, and improved tests can be further utilized to detect real-world bugs [43]. That is, we use artificial performance bugs to build and enhance the tests, and such tests could be used in the future to detect real-world performance bugs. If a performance test can detect an artificial performance bug, it is more likely to detect a real-world version of that bug.

6.4 Preliminary Analysis of PMT

To investigate the impacts of the five mutation operators, we devise a preliminary analysis based on five JMH benchmarks. We define the `hitting_count` as the total number of times that a benchmark executes (hits) a mutant statement. Furthermore, the `hitting_ratio` is the total number of times that a mutant statement is executed per second.

To investigate how large the effect of each mutation operator is when increasing `hitting_ratio`, we designed five JMH benchmarks. Each related benchmark is executed for 20 iterations against original source code and mutant version five times, yielding 100 data points containing measured throughput for one second. If differences between original results and mutant results increase, we assume that the effect of the performance bug is increased.

Figure 6 presents box plots of the five benchmarks' results for original and mutant executions, across increasing the `hitting_ratio`. To study the difference between the original microbenchmark execution and the microbenchmark executions after bug injections, we present the upper bound of estimated RCI in Table 2. The upper bound of RCI reflects the minimum performance degradation caused by bugs.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

Table 1: Examples of the five mutation operators.

Operator	Conditions	Example

PTW	Candidate expression should be a variable from any primitive types; i.e., `byte`, `boolean`, `short`, `int`, `long`, `float`, `double`, and `char`.	`- return this.count;`
		`+ return ((Long)(this.count)).longValue();`
		(a) Counter.java (Commit: `#8948b46`, `Eclipse-collections`)

STS	This operator should manipulate one method at a time to prevent propagation of changes.	`- StringBuilder result = new StringBuilder();`
		`+ StringBuffer result = new StringBuffer();`
		`...`
		`return result.toString();`
		(b) QueryRequest.java (Commit: `#4fb8d5a`, `Zipkin`)

EFL	Candidate loop should be a traditional `for`-loop that contains a loop counter.	`int i = 0;`
		`- for (;i < subscribers.length; i++) {`
		`+ for (Subscriber<? super R> o :subscribers){`
		`+ i = subscribers.getIndexOf(o);`
		`...}`
		(c) ParallelMap.java (Commit: `#0df952e`, `Rxjava`)

SOC	1) The left operand should be a variable or a `Boolean` literal.
	2) The right operand should be a method invocation that never return `Null`.
		`- if (done || emitter.isCancelled())`
		`+ if (emitter.isCancelled() || done)`
		(d) FlowableCreate.java (Commit: `#0df952`, `Rxjava`)

HWO	This pattern is not subject to any conditions.	`+ Thread.sleep(0,1);`
		`...`
		`return TreeBag.newBag();`
		(e) ImmutableEmptyBag.java (Commit: `#8948b46`, `Eclipse-collections`)

In general, for all five mutation operators, when we increased `hitting_ratio`, differences between original results and mutant results become more significant (the upper bound of RCI decreases). However, we also observed that the RCI upper bound of HWO (0.824) for 10^6 in Table 2 is greater than 10^5 (0.771), indicating that this conclusion may not be applicable to a very high `hitting_ratio` for a certain operator and the results may be limited by some external factors, such as overheads and variations occur due to hardware specific limitations. When `hitting_ratio` is lower than 10^3 times, differences are not significant in PTW, STS, and EFL with less than 1% performance degradation in all cases. On the other hand, in SOC and HWO, differences are significant in all `hitting_ratio`. In SOC and HWO, `hitting_count` is fixed while `hitting_ratio` is increased. This conveys that the effect of the two operators is high, even with one execution.

To summarize, when a benchmark executes a buggy statement (i.e., mutant statement) more times, the observed difference in results generally increases. In addition, differences are significant in all `hitting_ratio` in SOC and HWO.

7 Empirical Study

In this section, we describe our empirical study to test our hypothesis that the derived performance microbenchmarks from *ju2jmh* are preferable to using JUnit tests directly as performance proxies. Furthermore, we compare our derived microbenchmarks to human-written microbenchmarks to evaluate if the derived ones are superior than human-written ones. Moreover, we compare our derived microbenchmarks with those microbenchmarks automatically generated by AutoJMH [9] to evaluate if *ju2jmh* outperforms the existing framework.

Table 2: The upper bound of RCI for the five mutation operators with the increasing of `hitting_ratio`.

Operator	Hitting ratio						
	1	10	10^2	10^3	10^4	10^5	10^6
PTW	0.999	0.999	0.998	0.997	0.957	0.736	0.379
STS	1.0	1.0	0.998	0.990	0.932	0.959	0.687
EFL	0.997	1.0	0.998	0.998	0.986	0.967	0.920
SOC	0.843	0.848	0.847	0.847	0.854	0.840	0.826
HWO	0.878	0.877	0.864	0.820	0.800	0.771	0.824

7.1 Study Subjects

In this section, we introduce the three study subjects involved in this study. We experimented and evaluated our

[9] https://github.com/DIVERSIFY-project/autojmh-source-code

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

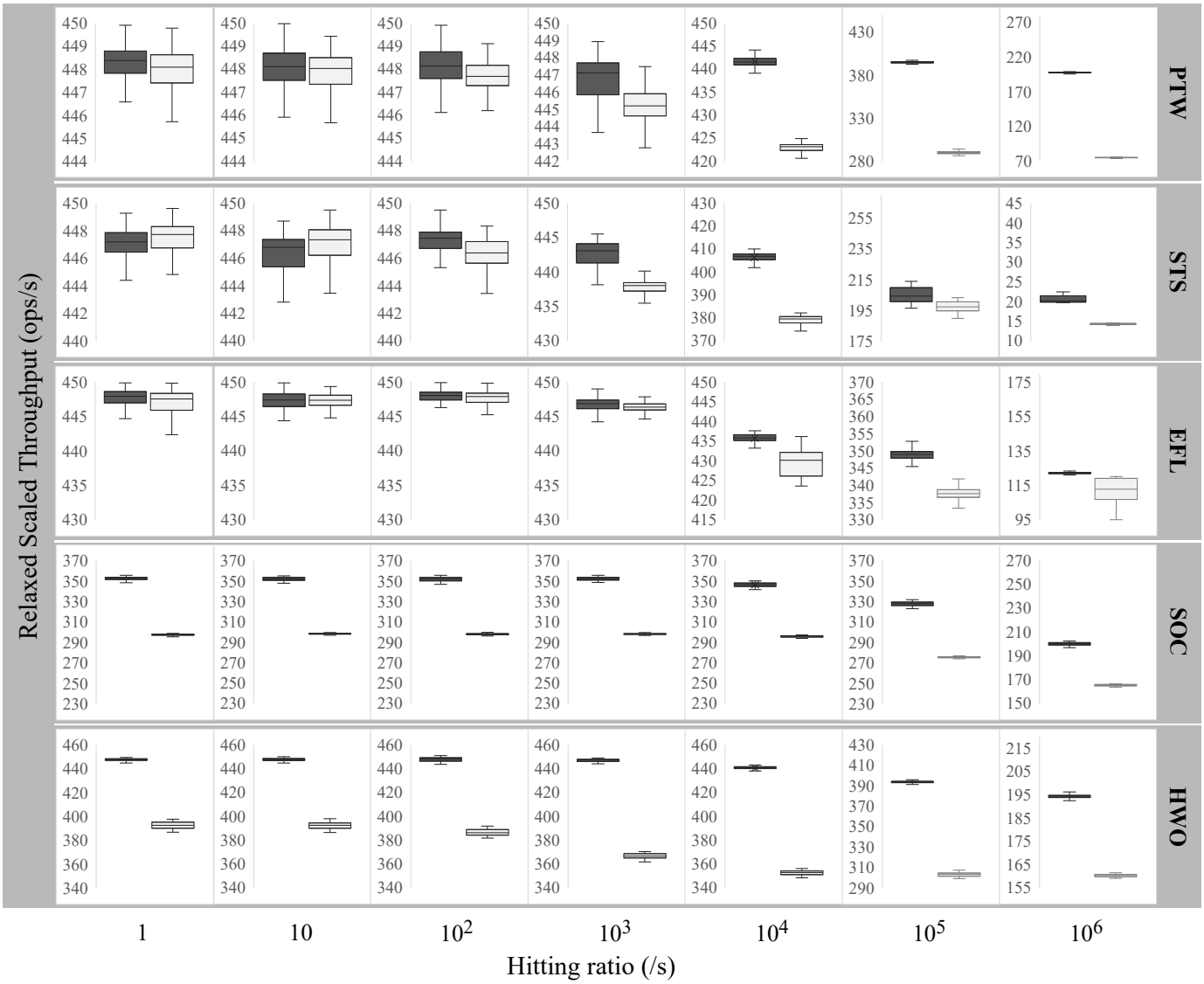


Figure 6: Preliminary analysis of the five mutation operators. Each of paired boxes contains obtained 100 data points for two cases of original source code executions and mutant executions, from each of five designed JMH benchmarks, across increasing `hitting_ratio`. Each box contains 100 data points obtained from a relevant benchmark. Black boxes represent data from original source code executions, and white boxes represent data from mutant executions.

two frameworks, i.e., *ju2jmh* and PMT, on three open-source Java projects, `Rxjava`, `Eclipse-collections`, and `Zipkin`, which have readily available JUnit test cases and JMH microbenchmarks. The selected three open-source projects are well-known, well-maintained, and were widely studied in prior studies on performance microbenchmarking [3], [5], [15], [17], [22], [37]. In addition, a recent study [5] lists the three subjects among the top 25 projects with the highest number of JMH micro-benchmarks.

Table 3 describes the detailed information of our studied subjects: the studied version ("Version") (i.e., the most recent version at the time our study began), the extracted metadata from **Github** including the numbers of stars (column **Stars**) and the number of contributors (column **Contr.**), the number of the source lines of code (column **SLOC**), the total number of JMH benchmarks (column **# JMH**) and selected JUnit test cases (column **# JUnit**).

Table 3: Overview of the studied subjects.

	Version	Stars	Contr.	SLOC	# JMH	# JUnit
RxJava	3	44.7k	277	311,975	1,217	9,825
Ec-collections	10.4.0	1.7k	88	135,017	986	24,758
ZipKin	2.7	14.4k	145	7,467	59	501

7.2 Experiment Settings

We deployed *ju2jmh* on three subjects' JUnit test suites to build $171,366$ *ju2jmh* benchmarks, of which $35,084$ were evaluated in this study, accounting for 48% of total $72,430$ tests evaluated. Our *PMT* framework analyzed a total of ∼454K SLOC and generated ∼149K artificial performance mutants, each of which was executed in a single position of the source code. During the validation procedure, ∼99% of the generated mutants were recognized as *possibly valid* mutants.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

Table 4: Overview of the selected mutants and covering tests that are involved in our experiment.

Mutation Operator	*RxJava*				*Eclipse-collections*				*Zipkin*			
	# Mutants	JMH	ju2jmh*	Auto.†	# Mutants	JMH	ju2jmh*	Auto.†	# Mutants	JMH	ju2jmh*	Auto.†
PTW	42	147	3074	42	18	121	418	18	30	21	119	30
STS	2	0	205	2	11	0	55	11	2	0	31	2
HWO	9	193	337	9	9	44	1226	9	2	6	21	2
EFL	8	121	573	8	8	0	174	8	6	11	33	6
SOC	5	0	10	5	3	8	132	3	0	0	0	0
Total	66	461	4199	66	49	173	2005	49	40	38	204	40

* equals to the number of JUnit tests
† AutoJMH benchmarks that is equal to the number of Mutants

`Eclipse-collections` includes a significant number of JUnit tests that take roughly four months to run once; as a result, we randomly selected ∼15% of them for examination while studying all tests from two other subjects. Afterward, in order to to have a fair and feasible evaluation approach, we selected a subset of the generated mutants with a wide variety of characteristics while considering all possible tests that cover them. To increase the chance of evaluating more tests against a specific mutant, we selected each mutant from the set of a class's mutants in different packages of the source code, as well as selecting mutants that are covered by more JMH benchmarks and *ju2jmh*/JUnit tests.

Table 4 provides an overview of the mutants and tests involved in our measurements. For each of five designed mutation operators and across the three study subjects, we present the number of selected mutants (column **# Mutants**), all contained JMH benchmarks (column **JMH**), *ju2jmh*/JUnit tests that cover any of the mutants (column **JUnit/ju2jmh**), and generated AutoJMH benchmarks (column **Auto.**).

Our study establishes three performance oracles: (1) If there is no source code update, the performance tests results should remain relatively unchanged; (2) If the existence of a performance bug is confirmed and covered by a performance test, there should be a variation in the testing results and such variation should be consistent with the characteristics of the performance bug; (3) If a mutation is injected and covered by a performance test, the testing results should vary, and the magnitude of the results should follow the same trend as the mutation triggered times.

7.3 Execution environment

To perform the execution procedure, we benefited from the power of cloud computing resources as a real-world environment. To have a consistent measurement process, for all the experiments in this study, we deployed c2-standard-4 (4 vCPUs, 16 GB memory) instances provided by *Google Cloud*[10]. Instances are run on Debian GNU/Linux 10 (buster) and OpenJDK 1.8. In total, our experiment consists of 99, 406 measurements, each containing 30 data points measured, i.e., each data point is the number of executions of one JMH microbenchmark case (annotated by `@benchmark`) in one second. It took ∼1656 machine hours to complete all the experiments in this study.

[10]https://cloud.google.com/

7.4 Results

We aim to answer three research questions. For each RQ, we present the motivation to answer the RQ, our approach to addressing the RQ, and the corresponding results.

RQ1: How stable are automatically generated performance microbenchmarks?

Motivation

A very first question arising while adopting a performance microbenchmarking is whether microbenchmarks (JMH, AutoJMH, JUnit, or microbenchmarks from *ju2jmh*) are stable enough to detect meaningful variations in microbenchmarking results. In recent studies, non-determinism has been identified as the main barrier to obtaining repeatable measurements in performance microbenchmarking. If a performance microbenchmark is not stable enough, it can report deceptive variations in microbenchmarking results when there are not any. Various tools and techniques aim at minimizing the effect of non-determinism at each level of abstraction [9], [44], [45]. JMH has recently become popular with achieving relatively stable performance results, while any form of using JUnit tests as benchmarks has been usually challenging and unstable. To evaluate the quality of performance microbenchmarks, we first measure their stability.

Approach

To answer RQ1, we studied the benchmarks in terms of their result variability. We executed all benchmarks of subjects individually in isolation. Each measurement consists of 30 iterations' result of a benchmark, similar to the number of trials in recent study [22].

To measure the stability of a benchmark, we have taken two steps: one encloses the stability in comparison with other benchmarks, while the other presents how soon a benchmark becomes stable (see Section 6.3 for more details).

(1) Benchmark's stability in comparison. In this step, for each measurement, we compute the relative standard deviation (RSD) across 30 iterations' results to statistically compare the benchmark's stability with others. We assume that benchmarks with RSD less than 1% are stable enough for detecting performance bugs [17], while benchmarks with RSD greater than 5% are unstable because it represents a relevant and clear variability in performance [38]. Due to the relatively large variation of benchmarks with RSD between 1% and 5%, they are not stable enough for

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

detecting small-size bugs, while they are still useful for detecting large-size bugs. However, most of the generated performance bugs do not have a large size and they might not be detected by benchmarks with RSD between 1% and 5%.

(2) Benchmarks' stability tendencies. If a performance benchmark reaches a stable stage sooner than others, it produces valuable results sooner. In this step, we devised a heuristic statistical strategy to determine which benchmark becomes stable first [37].

For each benchmark, we select i data points out of 30 iterations' result to form an original set ($P_{original_i}$), and similarly, we select another i different data points to form a microbenchmarking set ($P_{microbm_i}$). The selection size, i, ranges from 2 up to 15 data points ($i \in \{2, 3, ..., 15\}$). We repeat our selection strategy for 100 times, providing 100 pairs of $\{P_{original_i}, P_{microbm_i}\}$. Following prior study's approach [17] and by deploying *pa* [39], we estimate the confidence interval for the ratio of means with bootstrap [41], using 100 bootstrap iterations for each of 100 pairs (that resembles to the suggested 10,000 bootstrap iterations [42]). Therefore, **the width of estimated relative confidence interval** (RCIW) computed for each benchmark with i iterations, that indicates how a benchmark's results are spread and variable around the mean.

Results

(1) Benchmark stability in comparison. Figure 7 is a boxplot chart representing the distribution of RSD calculated (in %) for all benchmarks, across three testing frameworks and three subjects.

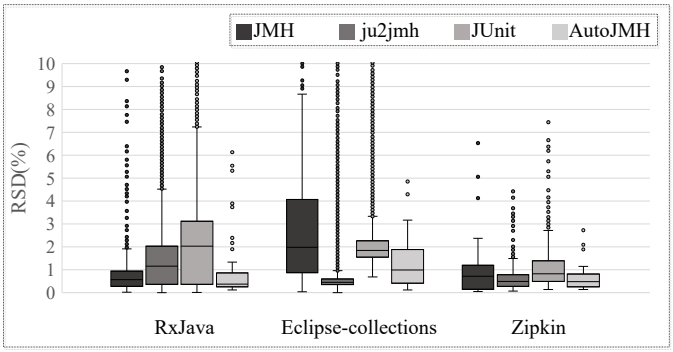


Figure 7: Distribution of the variability, using RSD (in %)

According to Figure 7, the box-and-whisker of RSD of *ju2jmh* benchmarks range from 0 % to 4.52% in `Rxjava` with a median of 1.16%, while the box-and-whisker of RSD of JUnit tests range from 0.01% to 7.24% with a median of 2.04% in the same subject. As we can see, JUnit tests have a higher median and maximum of RSD than JMH and AutoJMH benchmarks in `Rxjava`. `Zipkin` faces the same situation. For `Eclipse-collections`, JUnit tests have a higher median RSD (1.84%) than *ju2jmh* benchmarks (0.45%) and a higher maximum value in the box-and-whisker (3.34% vs. 0.96%), respectively. Because a larger RSD indicates that a benchmark is more unstable, these results imply that *ju2jmh* benchmarks are much more stable than JUnit tests in three studied subjects.

Likewise, we can also compare the stability of *ju2jmh* benchmarks and manually-written JMH benchmarks based on Figure 7. The median RSD of JMH benchmarks (0.57%) in `Rxjava` is smaller than *ju2jmh* benchmarks (1.16%), however, it is greater than *ju2jmh* benchmarks in `Eclipse-collections` (1.98% vs. 0.45%) and `Zipkin` (0.72% vs. 0.49%), as shown in this figure. If only the `Eclipse-collections` and `Zipkin` are considered, it appears that *ju2jmh* benchmarks are more stable than manually-written JMH benchmarks. However, when all three subjects are considered, we can only conclude that *ju2jmh* benchmarks beat JMH benchmarks in terms of stability in some subjects and not in others. As a result, *ju2jmh* benchmarks and manually-written JMH benchmarks are comparable.

Lastly, the stability of *ju2jmh* benchmarks can be also compared with AutoJMH benchmarks according to Figure 7. In terms of the median RSD, in `Rxjava`, AutoJMH benchmarks has a smaller value (0.37%) than *ju2jmh* benchmarks (1.16%). However, in `Eclipse-collections`, the median RSD of *ju2jmh* benchmarks (0.45%) is smaller than AutoJMH benchmarks (0.99%), and the box-and-whisker length of *ju2jmh* benchmarks (0.96%) is also smaller than AutoJMH benchmarks (3.05%), indicating that most of *ju2jmh* benchmarks are more stable than AutoJMH benchmark in this study subject. In `Zipkin`, the stability of most of *ju2jmh* benchmarks and most of AutoJMH benchmarks are comparable because of the close median RSD values (0.49% vs. 0.48%). Therefore, the stability of *ju2jmh* benchmarks are comparable to AutoJMH benchmarks based on the evaluation of these three subjects.

In conclusion, based on the comparisons between automatically generated *ju2jmh* benchmarks and the other three microbenchmarks, we can infer that, although *ju2jmh* leverages JUnit tests to generate microbenchmarks, the *ju2jmh* microbenchmarks are more stable than the original JUnit tests and their stabilities are comparable to manually-written JMH benchmarks and AutoJMH benchmarks.

In Table 5, we focus on how benchmarks stable and unstable are in detecting performance bugs. The table presents the proportion of benchmarks that are **stable** enough ($RSD \leq 1\%$) or **unstable** ($RSD \geq 5\%$).

Table 5: Stable and unstable benchmarks

Subject	Relative Standard Deviation							
	Stable (≤1%)				Unstable (≥5%)			
	JMH	ju2jmh	JUnit	Auto.†	JMH	ju2jmh	JUnit	Auto.†
RxJava	77.2%	43.8%	43.8%	81.1%	2.3%	3.9%	9.5%	7.5%
Ec-col.*	29.0%	91.5%	5.0%	51.0%	18.2%	1.7%	5.1%	6.1%
Zipkin	61.0%	84.6%	58.4%	82.5%	9.0%	0.0%	1.9%	0.0%

* Eclipse-collections
† AutoJMH

In `Eclipse-collections` and `Zipkin`, 91.5% and 84.6% of *ju2jmh* benchmarks are recognized stable, significantly higher than 5.0% and 58.4% for JUnit tests, 29.0% and 61.0% for JMH benchmarks, and 51.0% and 82.5% for AutoJMH benchmarks. In `Rxjava`, both *ju2jmh* and *JUnit* microbenchmarks produced equal number of stable benchmarks, but 9.5% of the JUnit tests are recognized unstable, higher than 3.9% for *ju2jmh* benchmarks. Despite the fact that JMH benchmarks (77.2%) and AutoJMH benchmarks (81.1%) are

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

more stable than *ju2jmh* benchmarks (43.8%), but in terms of unstable benchmarks, *ju2jmh* benchmarks and JMH benchmarks have very close values (3.9% vs. 2.3%) and *ju2jmh* benchmarks are better than AutoJMH benchmarks (3.9% vs. 7.5%). Based on these results, we can infer that *ju2jmh* benchmarks are generally more stable than JUnit tests and comparable to (or slightly more stable) manually-written JMH benchmarks and AutoJMH benchmarks.

Furthermore, we also check whether JUnit tests remain stable after being converted to *ju2jmh* benchmarks. In `Rxjava`, both *ju2jmh* and JUnit have the same amount of stable tests. Specifically, 86.4% of JUnit tests that recognized stable remain stable after being converted to *ju2jmh* benchmarks, 0.9% of JUnit tests that recognized unstable become stable, and 0.6% of JUnit tests that recognized stable become unstable. For the unstable JUnit tests that become stable, we rerun the associated *ju2jmh* benchmarks without warm-up iterations, and all of the new results achieve a higher RSD (%) ranging from 0.2% to 7.1% than the results with warm-up iterations, thus the test stability has deteriorated in all cases. As a result, warming up the system before running the benchmark is critical to achieve the stability of *ju2jmh* benchmarks in comparison to JUnit tests without warm-up iterations. For the stable JUnit tests that become unstable, we manually checked the source code and found that all of these tests contain tasks that deal with asynchronous executions, multiple threads or multiple processors, which are restricted in *ju2jmh* benchmarks. Moreover, all the JUnit tests in `Eclipse-collections` and `Zipkin` remain stable in form of *ju2jmh* benchmarks, none of JUnit tests that recognized unstable become stable, and none of JUnit tests that recognized stable become unstable. In conclusion, a high portion of JUnit tests remains stable after being converted to *ju2jmh* benchmarks.

(2) Benchmarks' stability tendencies. Figure 8 presents the results of the second step, which encloses how benchmarks grow more stable as the number of iterations increases. Every box represents the distribution of calculated RCIWs for all benchmarks of a testing framework with i iterations. The greater the RCIW, the more variation (less stability) there is in the result set. Accordingly, the larger the box-and-whisker, the more variation of stability there is between benchmarks of one type. For example, according to Figure 8, a benchmark usually produces more variable results through two iterations than a larger number of trials, such as 15 iterations. It is noted that, to facilitate a better comparison and clarification, we zoom in the chart so that a few whiskers and one box are shown out of the presenting range, but this does not affect the conclusion.

As illustrated in Figure 8, the median RCIW of JUnit tests is always greater than *ju2jmh*, JMH, and AutoJMH benchmarks, implying that JUnit tests are generally more unstable than the other three types of microbenchmarks. Furthermore, based on the following analysis, we can conclude that JMH, *ju2jmh*, and AutoJMH RCIWs are comparable among these three studied subjects. The median RCIW of *ju2jmh* benchmarks is always greater than JMH in `Rxjava` and `Zipkin`, but their differences are significantly smaller than the RCIW differences between JUnit tests and JMH/*ju2jmh* benchmarks, indicating that although *ju2jmh* benchmarks are more unstable than JMH benchmarks in these two subjects, their difference is very limited. `Rxjava` faces the same conclusions when *ju2jmh* is compared with AutoJMH. In addition, the median RCIW of *ju2jmh* benchmarks is lower than JMH and JUnit in `Eclipse-collections`, suggesting that those benchmarks are more unstable than *ju2jmh* benchmarks in this subject. Moreover, the median RCIW of *ju2jmh* benchmarks is always comparable with AutoJMH benchmarks in two subjects of `Eclipse-collections` and `Zipkin`. Lastly, in all three subjects, *ju2jmh* benchmarks' RCIWs are converging with JMH and AutoJMH benchmarks' RCIWs to a higher level of stability, with a very near level of stability after 15 iterations.

In conclusion, regarding stability assessing, both steps yield identical results: (1) in all three studied subjects, the generated *ju2jmh* benchmarks outperform JUnit tests; (2) *ju2jmh* and manually-written JMH benchmarks are comparable, but *ju2jmh* benchmarks outperform manually-written JMH benchmarks in two out of three studied subjects. (3) *ju2jmh* benchmarks are also comparable to AutoJMH benchmarks.

RQ1 Takeaway

Both methodologies of evaluating microbenchmark stability (using RSD and RCIW, respectively) reveal that although *ju2jmh* leverages JUnit tests to generate benchmarks, *ju2jmh* benchmarks significantly outperform JUnit tests in all studied subjects and are comparable to manually-written JMH benchmarks and AutoJMH benchmarks.

RQ2: Can performance microbenchmarks detect artificial performance bugs from mutation testing?

Motivation

While executing performance microbenchmarking, a performance bug can make developers question whether it comes from a noisy environment [37], [38], the benchmark's unstable quality [9], or even a bug. In RQ1, we assessed the stability of benchmarks. However, the stability itself is not sufficient for detecting performance bugs. A further evaluation of benchmarks is needed on their ability to detect performance bugs.

Approach

To address benchmarks' ability in detecting bugs, we leverage the *PMT* framework presented in Section 6 to inject artificial performance bugs into the source code. An artificial performance bug could be observed by a covering performance test by searching for degradations in its results. If the performance test can detect an artificial bug, it is more likely that the test can also detect a real-world similar performance bug.

First, to generate artificial performance bugs, we applied the *PMT* framework to the subject's source code. We build up a large number of systems versions, each containing one performance bug injected setting at a single source code location. Then, we extract all benchmarks that cover any of the selected mutants. Afterwards, we execute benchmarks that cover an injected performance bug, before and after

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

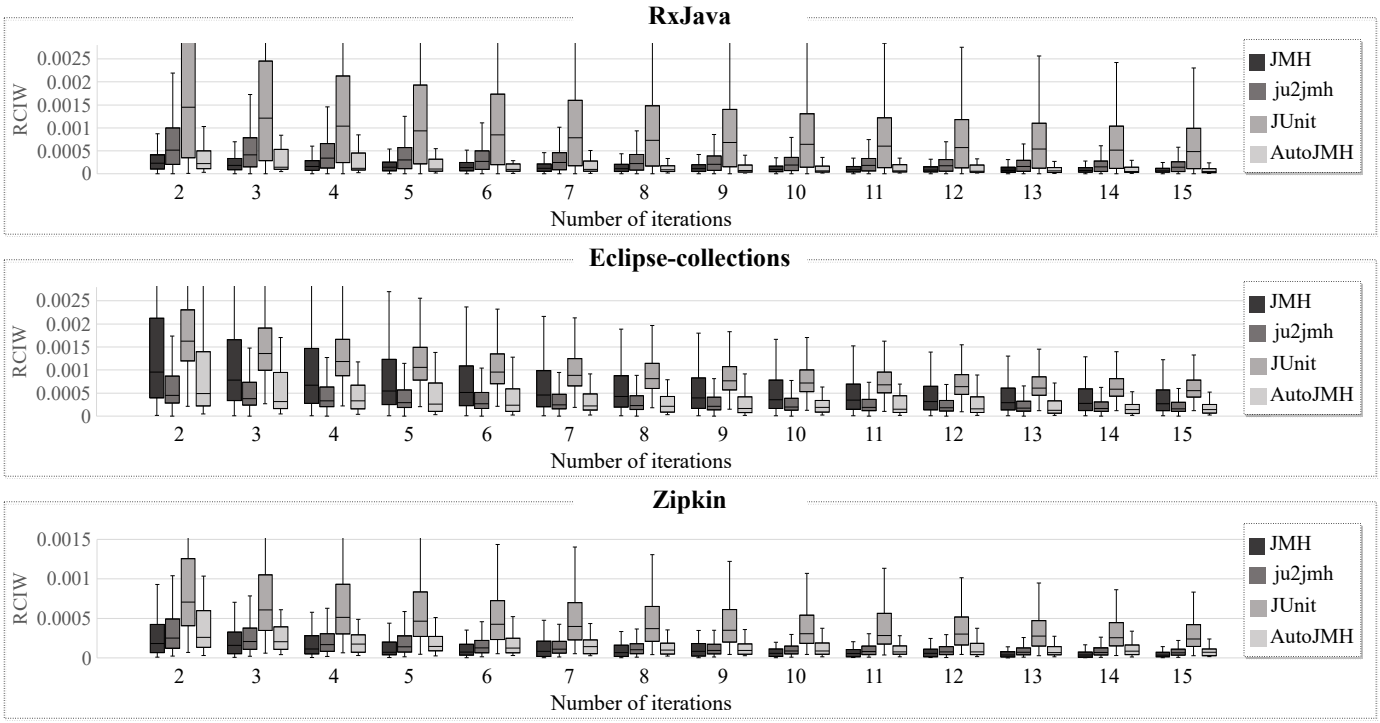


Figure 8: The box-plots of calculated RCIW for all benchmarks of a testing framework, ranging the iteration number i, $i \in \{2, 3, ..., 15\}$

injection, 30 times. Benchmarks are run in a sequence and through an isolated and controlled environment.

Relying on the goal of mutation testing in performance that is introducing performance bugs deliberately, we can look over the benchmarks' results to find any significant difference. Recent studies [3], [22] have advocated using *hypothesis statistical tests* (e.g., Wilcoxon rank-sum test) to label a significant difference in results as a performance bug, slowdown, or artificial bug. However, the prior study [37] concludes that testing with Wilcoxon rank-sum tests, due to high false-positive reported, is not a suitable vehicle for detecting performance degradation in cloud environments.

In order to accurately assess the difference between the microbenchmarking results before and after bug injection, we used mutation scores (i.e., the percentages of killed mutants defined in Section 6.3) to compare the analyzed subjects. We first assume that a benchmark could only kill a mutant if the throughput of the benchmark with the mutant is significantly lower than the throughput in the original system. For a studied benchmark, we estimate its **confidence interval for the ratios of means** (RCI), and the ratios are calculated based on the benchmark's throughputs (of before and after the bug injection) and by deploying bootstrap technique [40], [41], using 10,000 bootstrap iterations [42] with confidence level of 99%.

Therefore, the size of performance degradation caused by a performance bug can be defined as,

$$bug\_size = [1 - U_{RCI}]$$

that U_{RCI} is the upper bound of estimated RCI. The upper bound of estimated RCI, opposite to its lower bound, denotes the minimum bug occurring in the target system. We calculate the bug size in % and it reflects the effectiveness of mutant against the benchmark. We then assume **three different thresholds** as the minimum performance degradation that bugs produce, namely 1%, 5%, 10%, which has been used in prior study [22]. The thresholds are chosen from 1% to 10% because: (1) According to the preliminary analysis results of PMT in Table 2, bug size in the mutation operators SOC and HWO are always greater than 10%, which is why we set a 10% threshold for them. Although PTW, STS, and EFL mutation operators can/may achieve a bug size of 10% as the `hitting_ratio` increases by a large value, for consistency among different mutation operators, we still keep 10% as upper bound for all mutations. (2) In terms of selecting the lowest threshold, 1% is the lowest threshold used in the prior work [22], and the bugs with bug sizes less than 1% are difficult to detect. The three thresholds are chosen from 1% with a high falsely reported positive (bugs not caused by mutants) to 10% with a low falsely reported positive. If a bug size is greater than the threshold, there is a significant variation in results, which conveys that the benchmark could kill the mutant (detect the injected bug), and vice versa. The mutation score is then calculated to compare *ju2jmh* benchmarks to manually-written JMH benchmarks, JUnit tests, and AutoJMH benchmarks.

Results

Table 6 presents the percentage of tests that kill any of mutants and Table 7 presents the calculated mutation score over the three thresholds (columns **1%**, **5%** and **10%**), for the four testing frameworks (column **FW**) and the five mutation types (columns **PTW(%)**, **STS(%)**, **EFL(%)**, **SOC(%)** and **HWO(%)**).

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

Table 6: The percentage of tests that kill any of mutants from five operators, assuming that the mutant is killed if the *bug_size* $\geq$ 1%, 5%, and 10%.

Subject	FW	PTW (%)			STS (%)			EFL (%)			SOC (%)			HWO (%)			Total (%)		
		1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%
RxJava	JMH	34.6	6.1	0.6	-	-	-	37.1	8.2	3.3	-	-	-	55.4	41.4	35.7	44.0	21.4	16.0
	ju2jmh	29.2	3.0	0.6	40.0	12.1	2.9	44.4	6.4	2.7	20.0	0.0	0.0	56.3	42.1	39.4	34.0	7.0	4.1
	JUnit	6.5	0.7	0.1	21.4	4.3	0.9	10.8	3.3	0.3	0.0	0.0	0.0	42.1	39.4	38.5	10.6	4.3	3.2
	AutoJMH	64.2	11.9	0.0	100	100	100	50.0	25.0	12.5	20.0	0.0	0.0	66.7	44.4	33.3	64.7	19.6	11.8
Ec-col.*	JMH	21.4	8.2	4.1	-	-	-	-	-	-	0.0	0.0	0.0	54.5	50.0	50.0	28.9	18.4	15.6
	ju2jmh	31.5	8.3	1.6	33.9	9.4	5.6	37.3	12.6	6.8	43.1	11.3	3.7	37.0	9.2	4.8	37.1	9.4	4.3
	JUnit	20.3	5.5	1.1	35.8	11.3	7.5	36.7	13.2	6.8	40.9	10.6	3.7	18.2	7.3	4.2	25.2	7.8	3.9
	AutoJMH	66.7	16.7	5.6	54.5	18.1	9.1	62.5	25.0	12.5	100	66.7	33.3	66.7	66.7	66.7	66.7	33.3	21.2
Zipkin	JMH	19.0	4.7	0.0	-	-	-	9.0	9.0	9.0	-	-	-	33.3	33.3	33.3	18.4	7.8	5.2
	ju2jmh	12.6	2.5	0.8	25.8	16.1	0.0	30.3	0.0	0.0	-	-	-	47.6	33.3	33.3	21.0	7.3	3.9
	JUnit	11.7	2.5	0.8	25.8	16.1	0.0	0.0	0.0	0.0	-	-	-	42.8	33.3	33.3	16.1	7.3	3.9
	AutoJMH	36.7	10.0	3.3	100	100	0.0	50.0	0.0	0.0	-	-	-	100	100	100	43.2	16.2	8.1
Total	JMH	28.3	9.3	2.4	-	-	-	35.3	8.3	3.8	0.0	0.0	0.0	54.9	43.0	38.5	38.8	20.1	15.5
	ju2jmh	29.0	3.7	0.8	37.6	12.4	3.4	42.3	7.7	3.7	42.0	10.5	4.2	42.5	16.8	12.7	34.6	7.9	4.2
	JUnit	8.7	1.4	0.3	25.5	7.2	2.4	16.9	5.5	1.9	38.5	10.5	4.2	27.4	14.6	12.0	15.7	5.6	3.5
	AutoJMH	55.6	12.2	2.2	66.7	40.0	20.0	54.5	18.1	9.1	50.0	25.0	12.5	70.0	60.0	55.0	58.7	22.3	13.2
	Total (%)	19.5	2.7	0.5	32.1	10.2	2.9	30.2	6.6	2.7	38.9	10.3	3.6	32.9	15.0	11.7	25.8	7.4	4.4

* Eclipse-collections

Table 7: Mutation score of tests against the generated mutants from five operators, assuming that the mutant is killed if the *bug_size* $\geq$ 1%, 5%, and 10%.

Subject	FW	PTW (%)			STS (%)			EFL (%)			SOC (%)			HWO (%)			Total (%)			Coverage
		1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%	
RxJava	JMH	100	42.8	14.3	-	-	-	100	100	50.0	-	-	-	100	100	100	100	73.3	46.7	15 (22.7%)
	ju2jmh	95.2	50.0	16.7	100	100	100	100	87.5	75.0	40.0	0.0	0.0	100	88.9	88.9	92.4	57.6	34.8	66 (100%)
	JUnit	64.3	19.0	9.5	100	100	100	100	75.0	25.0	0.0	0.0	0.0	88.9	88.9	88.9	68.2	36.4	24.2	66 (100%)
	AutoJMH	64.2	11.9	0.0	100	100	100	50.0	25.0	12.5	20.0	0.0	0.0	66.7	44.4	33.3	60.6	19.7	9.1	66 (100%)
Ec-col.*	JMH	71.4	42.9	28.6	-	-	-	-	-	-	0.0	0.0	0.0	100	100	100	69.2	53.8	46.1	13 (26.6%)
	ju2jmh	94.4	61.1	22.2	81.8	36.4	27.3	62.5	62.5	50.0	100	100	100	100	77.8	66.7	87.8	61.2	40.8	49 (100%)
	JUnit	94.4	38.9	11.1	90.9	45.4	27.2	75.0	62.5	50.0	100	100	100	100	77.8	66.7	91.8	55.1	36.7	49 (100%)
	AutoJMH	66.7	16.7	5.6	54.5	18.1	9.1	62.5	25.0	12.5	100	66.7	33.3	66.7	66.7	66.7	65.3	30.6	20.4	49 (100%)
Zipkin	JMH	16.7	8.3	0.0	-	-	-	25.0	0.0	0.0	-	-	-	50.0	50.0	50.0	22.2	11.1	5.6	18 (45.0%)
	ju2jmh	40.0	10.0	3.3	100	100	0.0	66.7	0.0	0.0	-	-	-	100	100	100	50.0	17.5	7.5	40 (100%)
	JUnit	36.7	10.0	3.3	100	100	0.0	33.3	0.0	0.0	-	-	-	100	100	100	42.5	17.5	7.5	40 (100%)
	AutoJMH	36.7	10.0	3.3	100	100	0.0	50.0	0.0	0.0	-	-	-	100	100	100	45.0	17.5	7.5	40 (100%)
Total	JMH	53.8	26.9	11.5	-	-	-	62.5	50.0	25.0	0.0	0.0	0.0	90.0	90.0	90.0	60.9	43.5	30.4	46 (29.7%)
	ju2jmh	76.7	38.9	13.3	86.7	53.3	33.3	77.2	54.5	45.4	62.5	37.5	37.5	100	85.0	80.0	80.0	48.4	29.7	155 (100%)
	JUnit	61.1	20.0	7.8	93.3	60.0	33.3	72.7	50.0	27.2	37.5	37.5	37.5	95.0	85.0	80.0	69.0	37.4	23.9	155 (100%)
	AutoJMH	55.6	12.2	2.2	66.7	40.0	20.0	54.5	18.1	9.1	50.0	25.0	12.5	70.0	60.0	55.0	58.0	22.6	12.2	155 (100%)
	Total (%)	83.3	46.7	17.8	93.3	73.3	46.7	100	59.1	50.0	62.5	37.5	37.5	100	95.0	90.0	89.0	56.8	35.5	155 (100%)

* Eclipse-collections

Comparing *ju2jmh* benchmarks and JUnit tests: According to column **Total (%)** in Table 6, across the three subjects and for all three thresholds, more *ju2jmh* benchmarks kill a mutant than JUnit tests. In total, from 4.2% to 34.6% of all *ju2jmh* benchmarks kill a mutant, significantly higher than 3.5% to 15.7% for JUnit. Furthermore, over large degradations ($\geq$10%), low and equivalent percentage of *ju2jmh* and JUnit kill a mutant, while in small degradations ($\geq$1%), *ju2jmh* significantly outperforms JUnit.

Furthermore, according to column **Total (%)** in Table 7, *ju2jmh* achieves a higher mutation score than JUnit, with the exception of `Eclipse-collections` and the threshold of 1%. In total, *ju2jmh* achieves the mutation score from 29.7% to 80.0%, higher than JUnit with the mutation score of 23.9% to 69.0%. In `Eclipse-collections` and over the threshold of 1%, the difference between *ju2jmh* benchmarks and JUnit tests is slight (87.8 vs. 91.8). In other cases, *ju2jmh* outperforms JUnit.

In conclusion, while both *ju2jmh* and JUnit tests cover all of the mutants in the study, the percentage of *ju2jmh* benchmarks that kill a mutant is higher than JUnit tests, implying that for a given mutant operator, it is more likely to be killed by *ju2jmh* benchmarks than JUnit tests. In addition, *ju2jmh* benchmarks generally achieve higher mutation scores than JUnit tests. Therefore, *ju2jmh* is more effective in detecting performance bugs, regardless of the analysis perspective of mutation operators or tests.

Comparing *ju2jmh* and JMH benchmarks: We first compare percentage of *ju2jmh* and JMH benchmarks that kill a mutant using the analysis of column **Total (%)** in Table 6. In some cases, the percentage of *ju2jmh* benchmarks is greater than that of JMH benchmarks, while in other cases, it is lower. For example, 37.1% and 21.0% of *ju2jmh* benchmarks in `Eclipse-collections` and `Zipkin` kill a mutant with bug size $\geq$ 1%, higher than 28.9% and 18.4% of JMH benchmarks. However, in other cases, *ju2jmh* benchmarks achieve a smaller percentage.

We then compare the mutation scores of *ju2jmh* and

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

JMH benchmarks according to Table 7. Column **Total (%)** illustrates that *ju2jmh* benchmarks are more efficient than JMH benchmarks in killing mutants in general. On average, 29.7% to 80.0% of mutants are killed by *ju2jmh* benchmarks, higher than 30.4% to 60.9% by JMH benchmarks. In more than half of the cases, *ju2jmh* benchmarks achieve higher mutation scores than JMH benchmarks. Moreover, although manually-written JMH benchmarks are not far behind *ju2jmh* benchmarks in terms of mutation scores, they cover far fewer mutants (29.7% vs. 100%).

As presented in Table 6 and Table 7, although there are more percentages of JMH benchmarks that kill mutants than *ju2jmh* benchmarks, the overall coverage of all JMH benchmarks is much less than *ju2jmh*, implying that *ju2jmh* benchmarks have more coverage diversity, whereas manually-written JMH benchmarks are more specific to certain mutants with less coverage diversity. The manually-written JMH benchmarks would produce more accurate results if they could achieve higher coverage. However, this necessitates a great deal of manual effort, which is impractical in real-life development. Therefore, we can deduce that *ju2jmh* benchmarks cover more mutants than manually-written JMH benchmarks, and that the JMH benchmarks necessitate proper enhancements, such as higher mutant coverage, in order to achieve better results.

Comparing AutoJMH benchmarks with other tests: Similarly, when the percentage of AutoJMH benchmarks that kill a mutant is compared to those of other tests, AutoJMH benchmarks can have higher percentages than *ju2jmh* benchmarks, according to column **Total (%)** in Table 6. However, both benchmarks can generally achieve 100% mutant coverage, as presented in Table 7. In addition, AutoJMH benchmarks have limitations that prevent them from being trusted and valuable. First, according to Table 4, the number of generated AutoJMH benchmarks is significantly lower than the other tests, resulting in a limited data set for study. Second, AutoJMH is obsolete as we mentioned in Section 4. Third, AutoJMH is limited to manually configuring specific single statements. Such reasons limit the spread and application of AutoJMH.

Moreover, *ju2jmh* benchmarks achieve a higher mutation score than AutoJMH benchmarks in all cases across the three subjects and for all three thresholds, according to column **Total (%)** in Table 7. In total, *ju2jmh* achieves the mutation score of 29.7% to 80.0%, significantly higher than 12.2% to 58.0% for AutoJMH. In particular, in `Rxjava` and `Eclipse-collections`, *ju2jmh* benchmarks have mutation scores of at least 20.4% higher than AutoJMH benchmarks.

In conclusion, while both *ju2jmh* and AutoJMH benchmarks cover all mutants, *ju2jmh* benchmarks achieve higher mutation scores than AutoJMH benchmarks, indicating that *ju2jmh* benchmarks are more effective than AutoJMH benchmarks in detecting performance bugs.

Different effect of mutation operators' diversity: According to row **Total (%)** in Table 6, which presents the percentage of benchmarks that kill a mutant, across the threshold of 1%, 61.2% of JMH benchmarks, 65.4% of *ju2jmh* benchmarks, 84.3% of JUnit tests, and 41.3% of AutoJMH benchmarks failed in killing mutants. The highest percentage of benchmarks kill mutants from SOC (see the total mutation score in column **SOC**) with the percentage of 3.6% to 38.9%, and the smallest percentage of benchmarks kill mutants from PTW (column **PTW**) with the percentage of 0.5% to 19.5%. Moreover, mutants from HWO (column **HWO**) have a strong effect on benchmarks, particularly where 11.7% of benchmarks with a large bug size 10% is significantly greater than 0.5% of benchmarks that kill a mutant from PTW.

Table 7 yields the same finding, proving that different mutation operators have different effects on microbenchmarks. According to row **Total (%)**, which presents the calculated mutation score across all tests of the three study subjects, 35.5% to 89.0% of mutants are killed by the tests in total. The highest mutation score is achieved by HWO (column **HWO)**) where 90.0% to 100% of mutants are killed. The smallest mutation score is for SOC (column **SOC**) where 37.5% to 62.5% of mutants are killed. Such large variations in the results illustrate the diversity of mutant operators.

Exclusive mutation score: In order to provide more insight into *ju2jmh* benchmarks, we define the *exclusive mutation score*, which is the percentage of mutants that are exclusively killed by a framework and not by others. We calculate the exclusive mutation score for the four frameworks, across the three subjects, for the five mutation operators, and over the three thresholds. In total, over the threshold of 1%, *ju2jmh* benchmarks achieve the exclusive mutation score of 3.2%, higher than 2.2% for JMH benchmarks and 0.6% for JUnit tests, but less than 6.5% for AutoJMH benchmarks. Over the threshold of 5%, *ju2jmh* benchmarks achieve the exclusive mutation score of 9.0%, considerably higher than 2.2% for JMH, 0.6% for JUnit, and 6.5% for AutoJMH. Over the large threshold of 10%, *ju2jmh* benchmarks achieve the exclusive mutation score of 5.8%, higher than 0.6% for JUnit and 3.6% for AutoJMH, but less than 8.7% for JMH. In conclusion, most mutants are covered by multiple tests, with *ju2jmh* benchmarks covering more mutants exclusively than JUnit tests in all cases and JMH and AutoJMH benchmarks in the majority of cases.

RQ2 Takeaway

ju2jmh benchmarks are more effective in detecting performance bugs than JUnit tests and AutoJMH benchmarks. *ju2jmh* benchmarks cover more mutants than manually-written JMH benchmarks and JMH benchmarks necessitate proper enhancements, such as higher mutant overage, to achieve better results. Furthermore, we discover that different mutation operators can have various effects on benchmarks. In general, *ju2jmh* benchmarks can detect a higher proportion of mutants exclusively than other tests.

RQ3: Can performance microbenchmarks detect real-world performance bugs?

Motivation

One of the advantages of mutation testing is to find deficiencies in tests and improve them, thus improved tests can be further utilized for detecting real-world bugs [43]. We are motivated to know if benchmarks with higher performance mutation score can detect real-world performance bugs

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

better. On the other hand, we tend to know how similarly the generated mutants and real-world performance bugs behave in affecting a covering performance test.

Approach

To answer the motivation questions, we check whether a performance test is more likely to detect a similar real-world performance bug if it is able to detect a mutant.

For a specific real performance bug inside a system's source code, we first find all JUnit tests and JMH benchmarks that cover it. Next, we deploy *ju2jmh* to build benchmarks from JUnit tests and deploy the PMT tool to generate a mutant similar to the bug on the same source code location. Last, we execute covering *ju2jmh* and JMH benchmarks 30 times against the original system where the bug is fixed, the parent system that still contains the bug, and the system where the bug is replaced with a generated mutant. Then, we compare mutation results with real bug results. Accordingly, the tendency of performance tests in detecting a mutant is compared with the tendency of them in detecting real bugs.

Similar to previous research question, we predict the RCI for both the original system and the system containing real-world bugs (RCI_{bug}) and predict the RCI for the original system and its mutant version (RCI_{mutant}). Then, we compare both calculated RCIs to find similarities between them.

The real-world explored performance bugs are extracted from NFBugs dataset [23]. *NFBugs* is a dataset of 138 non-functional bug fixes in 67 open-source projects in Java or Python. NFBugs contains eight common performance bugs patterns from performance bugs that are already fixed by the projects' community. The building of our PMT tool is based on the real-world performance bugs collected from NFBugs dataset.

In total, there are 13 fixing of real-world performance bugs from 11 different projects in NFBugs dataset that are supported in our PMT tool (i.e., the bug can be reproduced by one of the five developed patterns). The 13 bug fixes are from three bug patterns of PTW, EFL, and STS. However, only one bug fix can be studied through this research question's experiment. For the rest of 12 bug fixes, in 11 of them, there is either no JMH/JUnit test in the system or there is no JMH/JUnit test that covers any of fixed bugs, and for the last bug fix, the corresponding system is no longer available in public.

The performance bug exists in `Storio`[11] and is fixed through commit of #566d3e9. There are 21 Junit tests in the system that cover the bug. Accordingly, we build 21 *ju2jmh* microbenchmarks and a mutant associated with the bug. Lastly, we compare the mutation score of *ju2jmh* benchmarks and JUnit tests.

Results

Figure 9 presents the calculated U_{RCI} for each of numbered 21 *ju2jmh* benchmarks and numbered 21 JUnit tests, against the mutant and the real bug. Similar to the previous research question, three thresholds (**1%**, **5%** and **10%**) are marked as three minimum *bug_size* that trigger the detection of bug. If calculated U_{RCI} is below each of the three thresholds' line, we conclude that the degradation is larger than the minimum *bug_size*. Table 8 summarizes Figure 9.

[11]https://github.com/pushtorefresh/storio.git

Table 8: Microbenchmarks that kill mutant and/or detect performance bug, assuming that the mutant is killed or the bug is detected if the *bug_size* $\geq$ 1%, 5%, and 10%.

ju2jmh						**JUnit**					
Mutant			Bug			Mutant			Bug		
1%	5%	10%	1%	5%	10%	1%	5%	10%	1%	5%	10%
66.7	42.8	4.7	66.7	42.8	0.0	61.9	19.0	0.0	61.9	23.8	0.0

According to Figure 9 and Table 8, *ju2jmh* appears to performs better than JUnit in either killing the mutant or detecting the real-world bug. For the three thresholds of bug sizes 1%, 5% and 10%, the percentage of *ju2jmh* benchmarks that kill the mutants is 4.7% (benchmark #1), 42.9% (benchmarks #1 to #9), and 66.7% (benchmarks #1 to #14) respectively, better than JUnit tests with the percentage of 0.0%, 19.0% (benchmarks #1 to #4), and 61.9% (benchmarks #1 to #13), indicating that *ju2jmh* benchmarks can detect the mutant better than JUnit tests. Similarly, *ju2jmh* performs better than JUnit in detecting real-world bugs. Over smaller degradations (1% and 5%), 66.6% and 42.8% of *ju2jmh* microbenchmarks detect the presence of the bug, higher than 61.9% and 23.8% for JUnit tests. None of the *ju2jmh* or JUnit tests can detect real-world bugs with a larger bug size threshold (10%). As a result, *ju2jmh* performs better than JUnit in both killing the mutant and detecting the bug.

According to Figure 9, both *ju2jmh* benchmarks and JUnit tests produce results with a similar distribution against the mutant and the bug. On average, the differences of calculated U_{RCI} between the mutant and the bug is 0.36% for *ju2jmh* benchmarks and 0.26% for JUnit tests, enclosing the similarities between results of benchmarks against the mutant and results of benchmarks against the bug. However, in all cases, the tendency of *ju2jmh* benchmarks in killing the mutant or detecting the bug is higher than JUnit tests. *ju2jmh* benchmarks have a lower calculated U_{RCI} value (higher tendency) than JUnit tests from 0.3% to 2.98% in killing the mutant, and from 0.23% to 2.63% in detecting the bug. In conclusion, the distribution of *ju2jmh* benchmarks' results and JUnit test results is similar, but the tendency of *ju2jmh* benchmarks in both killing the mutant and detecting the bug is higher than JUnit tests.

RQ3 Takeaway

According to our experiment of a real-world performance bug and a similar generated mutant, *ju2jmh* performs better than JUnit in killing the mutant and detecting the bug. Furthermore, the mutant and real-world bug affect benchmarks in a very similar way.

RQ4: What are the major factors affecting a microbenchmark's ability to detect performance bugs?

Motivation

To determine a performance microbenchmark is effective or unsatisfactory, there are a number of metrics that require

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

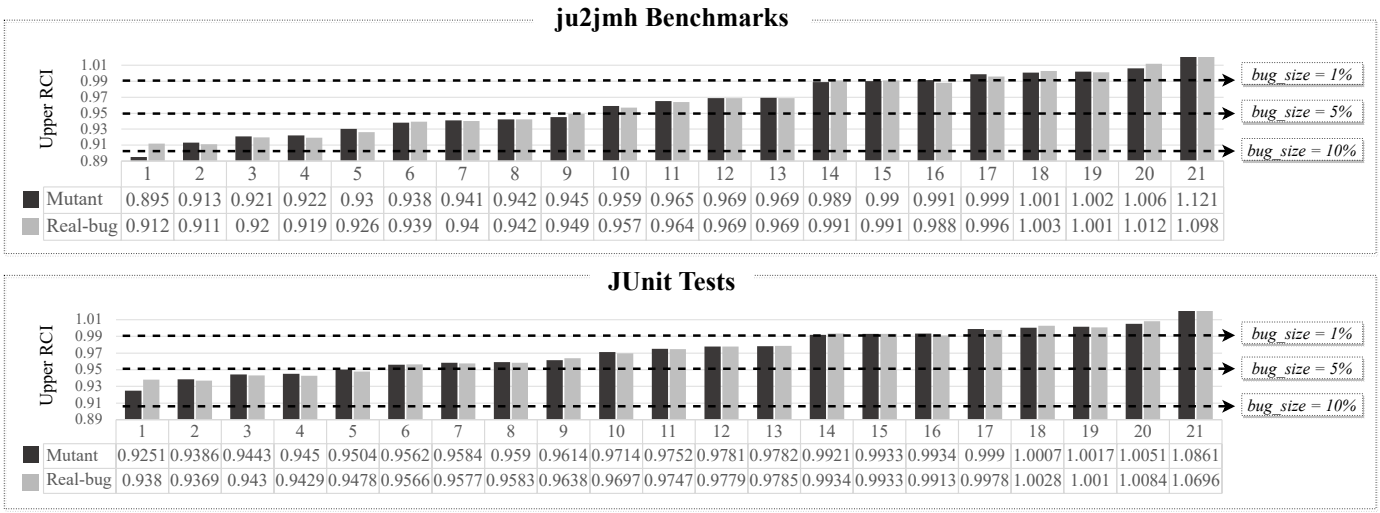


Figure 9: The calculated U_{RCI} for each of 21 *ju2jmh* benchmarks and 21 JUnit tests, against the mutant and the real bug.

further investigation and comparisons. In this section, we highlight three causes that significantly affect the ability of microbenchmarks to detect bugs. The causes **c1** and **c3** in the following are derived from prior research [8]. We examine three of the eight causes they identified and merge two of them into **c1**. In addition, **c2** is derived based on both the prior work [3] and the aforementioned analysis of RQ results.

c1. Too low workload: performance bugs could not be detected in benchmarks with a modest workload of payloads. In other words, most of mutants are killed by the benchmarks with relatively large workload.

Ding *et al.* [8] point out that not enough microbenchmark execution repetition could have a significant impact on a performance microbenchmarks' ability to find bugs. During a microbenchmark execution, if we raise the number of iterations of microbenchmark code, the workload can be increased without exceeding the hardware limit. As a result, we have combined these two reasons into one.

c2. Unstable microbenchmarks: Laaber and Leitner [3] reveal that the variability of a benchmark has an impact on benchmarking results. Furthermore, both RQ1 and RQ2 indicate that *ju2jmh* benchmarks outperforms JUnit tests in all studied subjects and is comparable to JMH benchmarks when two separate aspects are taken into account (i.e., the microbenchmark stabilities and microbenchmark ability to detect bugs). The same conclusions may imply that microbenchmark stability affects microbenchmark ability to detect bugs, which could be of interest.

c3. Limited mutant coverage: performance bugs are generated by the *PMT* framework, however they only affect one point in the source code (mostly one line of source code). Therefore, to investigate this cause, we considered the total number of times that the source code line containing bugs (mutants) was hit during the microbenchmark execution as a coverage metric. We claim that the more times a benchmark hits the buggy line, the more significant performance degradation in microbenchmarking results exists.

Other causes: there are other known/unknown causes for future study that might impact benchmarks in exposing bugs. For example, benchmarks may access to IO/network/resource [8], natural internal and external noises [37], [38], ideal during execution [8], and the design of the benchmark itself [5]. These other causes are not evaluated here, and are left for future research.

Approach

To answer RQ4, we examined each of the three aforementioned causes against all of the RQ2's benchmark tests that killed a mutant. Since there were not many benchmarks that killed a mutant with the bug size $\geq$ 5% and 10%, we only studied on the threshold of 1% as the minimum bug size.

To examine workload **(c1)**, we take the *throughput (ops/s)* of benchmarks as the metric. In addition, we take the *RSD (%)* of benchmarks and the *hitting_count (/s)* to study on the second **(c2)** and third **(c3)** causes respectively. For each metric (i.e., *throughput/RSD/hitting_ratio*), we calculate the number of benchmarks that could kill the mutant with different values of metrics for subsequent analysis. Specifically, we split the range of all measurements (of a testing framework and across all five mutant types) into four equal-size groups from the *minimum* value to the *maximum* value in the data-set. Then, we count the total number of benchmarks that kill a mutant in each group.

In prior RQs, we compare *ju2jmh* to other testing frameworks and find that *ju2jmh* outperform other testing frameworks in general. Therefore, the question of why *ju2jmh* outperforms other testing frameworks arises. The *ju2jmh* benchmarks achieve 100% mutant coverage, which means that each mutant must be executed. For mutants merely covered by *ju2jmh* benchmarks, it is obvious that *ju2jmh* benchmarks perform better due to mutant coverage. For mutants covered by multiple tests, we perform a study on them. We first select all mutants that are covered by all four types of tests. We choose the top mutant cases where *ju2jmh* benchmarks outperform other tests the most and conduct manual analysis on them. For each of the five mutation types, we extract a ranking list of the mutants for which *ju2jmh* outperforms the other three frameworks in terms of killing the mutants. The ranking is determined by the largest

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

Table 9: The number of tests that killed any of mutants, according to the three causes with four groups of the related metric, for the three testing frameworks, and across three studied subjects.

Subject	Framework	c1: too low workload				c2: unstable tests				c3: limited coverage				Total
		g1	g2	g3	g4	g1	g2	g3	g4	g1	g2	g3	g4	
RxJava	JMH	38	50	82	33	195	2	3	3	29	52	91	31	203
	ju2jmh	32	115	193	1,088	1,328	83	7	4	79	870	424	55	1,428
	JUnit	14	34	100	319	453	8	3	3	18	244	160	45	467
	AutoJMH	3	1	4	32	35	1	0	4	3	2	28	7	40
Ec-col.*	JMH	23	23	2	2	40	5	4	1	2	22	18	8	50
	ju2jmh	10	15	63	653	717	18	5	1	5	390	335	11	741
	JUnit	9	10	42	441	485	12	4	1	4	149	336	13	502
	AutoJMH	6	0	2	24	31	0	0	1	14	11	4	3	32
Zipkin	JMH	6	0	0	1	5	0	1	1	4	2	0	1	7
	ju2jmh	1	4	9	29	39	3	0	1	2	12	12	17	43
	JUnit	1	1	12	19	31	1	0	1	6	5	9	13	33
	AutoJMH	2	0	6	10	11	5	0	2	1	7	4	6	18
Total		145 (4.1%)	253 (7.1%)	461 (12.9%)	2,651 (74.4%)	3,370 (94.6%)	138 (3.9%)	27 (0.8%)	23 (0.6%)	167 (4.7%)	1,766 (49.6%)	1,421 (39.9%)	210 (5.9%)	3,564

* Eclipse-collections

bug_size difference between *ju2jmh* benchmarks and the tests from the other three frameworks. If a mutant is covered by multiple tests that are specific to a type, we choose the best result as a representation of this test type. Then top five mutants with the largest *bug_size* difference between *ju2jmh* benchmarks and the other tests are then selected. Lastly, we manually inspect the top five cases, such as static analysis, to determine why *ju2jmh* benchmarks outperform other tests.

Similarly, we follow the procedure from the manual analysis above to study the reasons why the mutants can only be killed by *ju2jmh* benchmarks but not by other tests. We first extract all mutants that can only be killed by *ju2jmh* benchmarks with three various thresholds for throughput reduction (1%, 5% and 10%). For each threshold and mutation operator type, after ranking *ju2jmh* benchmarks by comparing the maximum difference of *bug_size* between them and the other tests, we choose the top cases to perform the manual analysis.

Results

Table 9 presents the number of microbenchmarks that killed a mutant according to each of four groups of the three causes.

c1. Too low workload:

In total, 74.4% of mutants are killed by benchmarks with the largest workload (column **c1.g4**), significantly higher than three other groups. In all cases of column **c1**, *ju2jmh*, JUnit, and AutoJMH significantly confirms our claim that benchmarks with the largest workload are better in detecting bugs, while JMH is against our claim.

In `Rxjava`, 1,088 (76.2%) *ju2jmh* benchmarks, 319 (68.3%) JUnit tests, and 32 (80.0%) AutoJMH benchmarks belong to the largest workload group (**g4**), while JMH benchmarks are normally distributed among four groups and the largest workload's group contain only 16.3% of the benchmarks. Furthermore, in `Eclipse-collections` and `Zipkin`, similar findings are obtained in comparing *ju2jmh*, JUnit, and AutoJMH. However, JMH completely stands against our claim in these two subjects where 23 (46%) `Eclipse-collections` benchmarks and 6 (85.7%) `Zipkin` benchmarks belong to the smallest workload's groups (**g1**).

In conclusion, *ju2jmh*, JUnit, and AutoJMH all confirm that benchmarks with the largest workload are better in detecting performance bugs, but JMH does not necessarily support this claim.

c2. Unstable microbenchmarks:

Column **c2** of table 9 depicts a highly significant effect on stability, with the most stable benchmarks (column **c2.g1**) killing 94.6% of mutants. All four test frameworks confirm that most of mutants are killed by the most stable benchmarks. Specifically, across the three subjects, 71.4% to 96.1% of JMH benchmarks, 90.7% to 96.8% of *ju2jmh* benchmarks, 93.9% to 97.0% of JUnit tests, and 61.1% to 96.9% of AutoJMH benchmarks belong to the most stable benchmarks that killed a mutant.

In conclusion, microbenchmark stability has a significant impact on detecting performance bug. The more stable benchmarks can better detect performance bugs.

c3. Limited mutant coverage:

In general, results do not necessarily confirm that benchmarks need higher `hitting_ratio` (e.g., column **c3.g4**) to detect performance bugs. However, if the benchmark hit the buggy statement with lower ratio (e.g., column **c3.g1**), the mutant could not be killed by the benchmark. In total, two middle groups (**c3.g2** and **c3.g3** killed 89.4% of all mutants, and only 4.7% of mutants are killed in a situation with the lowest `hitting_ratio` (**c3.g1**). As a result, we can conclude that in order to detect performance bugs, microbenchmarks require appropriate `hitting_ratio` values (not too high or too low). In other words, mutants with too low `hitting_ratio` cannot sufficiently degrade microbenchmarks. Moreover, microbenchmarks rarely hit a mutant with a large `hitting_ratio` to make significant degradations. The data trend indicating the number of microbenchmarks detecting bugs decreases in an interval with a high `hitting_ratio` reinforces the view stated in section 6.4, implying that some external factors, such as hardware limitations, may have an impact on microbenchmarks results.

According to Table 9, we can conclude that the positive effect of each of the three discussed causes can result in better detection of the performance bug. We conclude that benchmarks with sufficient workload, higher stability, and

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

appropriate coverage can better kill the mutant. The three approaches of *ju2jmh*, JUnit, and AutoJMH confirm our claim that benchmarks with the largest workload can better detect bugs, while JMH does not necessarily confirm it. In all cases, all four approaches confirm that the most stable benchmarks can better detect bugs. Lastly, results do not confirm that benchmarks need high coverage, but they need an appropriate coverage of the bug to be able to detect the bug.

Our manual study identified three reasons why *ju2jmh* benchmark outperforms others. In particular, the experiment yields five top-five cases (out of a total of 25) in which the *ju2jmh* benchmark outperforms other tests the most based on *bug_size*. The maximum *bug_size* difference between *ju2jmh* benchmarks and other tests ranges from 0.5% to 97.2%. Among the 25 cases, AutoJMH benchmarks are involved in the majority of cases (16 of 25), while JMH benchmarks are involved in the fewest (4 of 25 cases). In addition, we manually analyze another 27 cases where mutants that can only be killed by *ju2jmh*. Based on the total 52 (i.e., $25+27$) cases, we found the reasons why *ju2jmh* benchmarks outperform other tests, which can be summarized as follows: (1) In 28 cases, *ju2jmh* benchmarks hit the mutants more often than the other frameworks (more than $4\times$ to $400,000\times$); (2) In 22 cases, *ju2jmh* benchmarks have higher stability (lower RSD) than the other frameworks; (3) In 13 cases, *ju2jmh* benchmarks have a more proper workload size (not too high or too low) than the other frameworks. The detailed experiment results are attached to Table 10 in the Appendix section.

In summary, to answer this RQ4, we used prior work to derive three causes and conducted three experiments to measure these three causes. The results indicate that (1) Too low workload can prevent *ju2jmh* benchmarks from detecting performance bugs effectively. The same conclusion applies to JUnit tests and AutoJMH benchmarks, but not to JMH benchmarks. (2) Unstable microbenchmarks can also impede microbenchmarks from detecting bugs, and this finding is not limited to a particular testing framework. (3) Performance bugs are more likely to be detected in microbenchmarks with appropriate execution times for bugs during the microbenchmarking period. Too low or too high execution times can cause microbenchmarks to fail to detect performance bugs.

RQ4 Takeaway

To answer this RQ, we exposed three causes that can prevent microbenchmarks from detecting performance bugs effectively: (1) Too low workload (this applies to generated *ju2jmh*, JUnit and AutoJMH tests, but not to manually-written JMH benchmarks). (2) Unstable benchmarks. (3) Limited mutant coverage as a result of inappropriate execution times (neither too low nor too high).

8 Threats to Validity

In this section, we discuss the threats to the validity of our study.

8.1 External Validity

We performed our evaluations on three popular open-source Java software systems from a limited system set that extensively deploys JMH along with JUnit. However, the subjects' JMH benchmarks are actively maintained by professional development teams and include sufficient microbenchmarks, thus they are suitable to be used in our experiments.

We developed five mutation operators in our *PMT* framework, but there would be many other known/unknown performance bug patterns in real-world systems. We randomly selected our study mutants from a broad set of generated mutants. Various mutants can have a wide range of characteristics that our chosen mutants may not cover. In the future, we will explore more mutants and extend our framework to support more mutant types.

During our study, we found that some JUnit tests are performance unit tests rather than functional unit tests. At the time of writing, our tool does not distinguish them since it is tricky to do so through static analysis of source code. Future work may consider addressing this issue to improve our existing approach.

8.2 Internal Validity

Monitoring performance is always challenging with noises [9]. To minimize such errors as much as possible, (1) for each measurement, we executed benchmarks for 30 iterations each, (2) in isolated, controlled, and large-scale cloud computing resources provided by *Google Cloud*, and (3) we reduced random bias by performing an evaluation strategy, i.e., bootstrapping.

Performance testing is vulnerable to environmental noises and running performance tests on virtual computing instances might affect the reliability of results more than on bare-metal computing servers. However, bare-metal servers are more expensive than cloud environments, and the data noise produced by cloud-based studies is usually acceptable. There exist many prior studies [3], [8], [22], [28], [37], [38] that have been conducted on cloud environments for performance testing. In addition, although the variability of cloud environment can affect statistical analysis, we repeat each test 30 times to mitigate data noise caused by such variability.

9 Conclusion

The goal of performance microbenchmarking is to catch degradations (e.g., slowdowns) as early as possible, such as checking (and microbenchmarking) every build of the system. In this paper, we look into how to help with microbenchmark construction and quality assurance. On the one hand, our *ju2jmh* tool can automatically construct ready-to-execute JMH benchmarks from a wide set of readily available JUnit tests, that potentially relieve developers from designing challenges and costs. One the other hand, we implement a *PMT* framework to automatically assess the quality of microbenchmarks in detecting bugs.

To further assess the quality of our automatically generated *ju2jmh* benchmarks, we devise a study to compare the *ju2jmh* benchmarks to manually-written JMH benchmarks

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

and using JUnit tests directly as performance proxies in terms of microbenchmark stability and ability of performance bug detection. For microbenchmark stability, generated *ju2jmh* benchmarks have been proven to be comparable with manually-written JMH benchmarks, while they significantly outperform JUnit tests in all analyzed subjects. For microbenchmark efficiency, we can similarly conclude that *ju2jmh* benchmarks outperform JUnit benchmarks and are comparable with manually-written JMH benchmarks. However, *ju2jmh* benchmarks are able to cover more of the software applications than JMH benchmarks during the microbenchmarking execution. In addition, when compared to developing human-written JMH benchmarks, our automated tool can save developers time and effort in microbenchmark construction.

In future, we aim to extend the *ju2jmh* by handling two common noises of dead code elimination and constant folding [16]. Furthermore, our *PMT* framework is developed with five operators, while there would many other performance bug patterns in reality. In future works, we aim to advance the *PMT* with more mutation operators. In addition, we would like to investigate and evaluate other factors that may affect microbenchmarks' ability to detect performance bugs.

REFERENCES

[1] E. Weyuker and F. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147–1156, 2000.

[2] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015. DOI: 10.1109/TSE.2015.2445340.

[3] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 119–130, ISBN: 9781450357166. DOI: 10.1145/3196398.3196407.

[4] A. Shipilev, *Nanotrusting the nanotime.* Accessed: 2021-07-02, 2021. [Online]. Available: https://shipilev.net/blog/2014/nanotrusting-nanotime/.

[5] D. E. Damasceno Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019. DOI: 10.1109/TSE.2019.2925345.

[6] L. Bulej, T. Bures, V. Horký, *et al.*, "Unit testing performance with stochastic performance logic," *Automated Software Engineering*, vol. 24, pp. 139–187, 2015.

[7] C.-P. Bezemer, S. Eismann, V. Ferme, *et al.*, "How is performance addressed in DevOps?" In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19, Mumbai, India: Association for Computing Machinery, 2019, pp. 45–50, ISBN: 9781450362399. DOI: 10.1145/3297663.3309672.

[8] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?" In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1435–1446, ISBN: 9781450371216. DOI: 10.1145/3377811.3380351.

[9] T. Kalibera and R. Jones, *Quantifying performance changes with effect size confidence intervals*, Accessed: 2021-07-02, 2021. arXiv: 2007.10899 [stat.ME].

[10] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" *SIGPLAN Not.*, vol. 44, no. 3, pp. 265–276, Mar. 2009, ISSN: 0362-1340. DOI: 10.1145/1508284.1508275.

[11] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, no. 10, pp. 57–76, Oct. 2007, ISSN: 0362-1340. DOI: 10.1145/1297105.1297033.

[12] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 60–71, ISBN: 9781450327565. DOI: 10.1145/2568225.2568232.

[13] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 237–246. DOI: 10.1109/MSR.2013.6624035.

[14] P. Stefan, V. Horky, L. Bulej, and P. Tuma, "Unit testing performance in java projects: Are we there yet?" In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 401–412, ISBN: 9781450344043. DOI: 10.1145/3030207.3030226.

[15] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 373–384, ISBN: 9781450344043. DOI: 10.1145/3030207.3030213.

[16] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, ser. ASE 2016:Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, Sep. 2016.

[17] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 989–1001, ISBN: 9781450370431. DOI: 10.1145/3368089.3409683.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

[18] H. M. AlGhamdi, C.-P. Bezemer, W. Shang, *et al.*, "Towards reducing the time needed for load testing," *Journal of Software: Evolution and Process*, e2276, 2015, e2276 smr.2276. DOI: https://doi.org/10.1002/smr.2276. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2276.

[19] H. M. Alghmadi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 279–289. DOI: 10.1109/ICSME.2016.46.

[20] A. Shipilev, *Java benchmarking as easy as two timestamps.* Accessed: 2021-07-02, 2021. [Online]. Available: http://shipilev.net/blog/2014/nanotrusting-nanotime.

[21] C. Click, *The art of java benchmarking*, Accessed: 2021-07-02, 2010. [Online]. Available: http://www.azulsystems.com/presentations/art-of-java-benchmarking.

[22] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Software Testing, Verification and Reliability*, vol. 31, no. 5, e1728, 2021, e1728 stvr.1728. DOI: https://doi.org/10.1002/stvr.1728. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1728.

[23] A. Radu and S. Nadi, "A dataset of non-functional bugs," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 399–403. DOI: 10.1109/MSR.2019.00066.

[24] Google Corporation, *Caliper*, Accessed: 2021-07-02, 2015. [Online]. Available: https://github.com/google/caliper.

[25] O. Corporation. (2016). "Java microbenchmarking harness (jmh)." Accessed: 2021-07-02, [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/.

[26] I. Clarkware Consulting, *Junitperf*, Accessed: 2021-07-02, 2009. [Online]. Available: https://github.com/noconnor/JUnitPerf.

[27] M. Papadakis, M. Kintis, J. Zhang, *et al.*, "Chapter six - mutation testing advances: An analysis and survey," in, ser. Advances in Computers, A. M. Memon, Ed., vol. 112, Elsevier, 2019, pp. 275–378. DOI: https://doi.org/10.1016/bs.adcom.2018.03.015.

[28] C. Laaber, "Continuous software performance assessment: Detecting performance problems of software libraries on every build," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 410–414, ISBN: 9781450362245. DOI: 10.1145/3293882.3338982.

[29] G. Jin, L. Song, X. Shi, *et al.*, "Understanding and detecting real-world performance bugs," *SIGPLAN Not.*, vol. 47, no. 6, pp. 77–88, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254075.

[30] S. Zaman, B. Adams, and A. E. Hassan, "A qualitative study on performance bugs," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR '12, Zurich, Switzerland: IEEE Press, 2012, pp. 199–208, ISBN: 9781467317610.

[31] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009, ISSN: 1382-3256. DOI: 10.1007/s10664-008-9077-5.

[32] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 1013–1024, ISBN: 9781450327565. DOI: 10.1145/2568225.2568229.

[33] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," *SIGPLAN Not.*, vol. 50, no. 6, pp. 369–378, Jun. 2015, ISSN: 0362-1340. DOI: 10.1145/2813885.2737966.

[34] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: Adaptive selection of collections," *SIGPLAN Not.*, vol. 44, no. 6, pp. 408–418, Jun. 2009, ISSN: 0362-1340. DOI: 10.1145/1543135.1542522.

[35] M. Linares-Vásquez, G. Bavota, M. Tufano, *et al.*, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 233–244, ISBN: 9781450351058. DOI: 10.1145/3106237.3106275.

[36] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 902–912. DOI: 10.1109/ICSE.2015.100.

[37] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empirical Softw. Engg.*, vol. 24, no. 4, pp. 2469–2508, Aug. 2019, ISSN: 1382-3256. DOI: 10.1007/s10664-019-09681-1.

[38] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, Apr. 2016, ISSN: 1533-5399. DOI: 10.1145/2885497.

[39] C. Laaber, *Pa - performance (change) analysis using bootstrap*, Accessed: 2021-07-02, 2021. [Online]. Available: https://github.com/chrstphlbr/pa.

[40] S. Ren, H. Lai, W. Tong, *et al.*, "Nonparametric bootstrapping for hierarchical data," *Journal of Applied Statistics*, vol. 37, no. 9, pp. 1487–1498, 2010. DOI: 10.1080/02664760903046102. eprint: https://doi.org/10.1080/02664760903046102.

[41] A. C. Davison and D. V. Hinkley, *Bootstrap Methods and their Application*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. DOI: 10.1017/CBO9780511802843.

[42] T. C. Hesterberg, "What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum," *The American Statistician*, vol. 69, no. 4, pp. 371–386, 2015, PMID: 27019512. DOI: 10.1080/00031305.2015.1089789. eprint: https://doi.org/10.1080/00031305.2015.1089789.

[43] G. Petrovic, M. Ivankovic, G. Fraser, and R. Just, "Does mutation testing improve testing practices?"

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.

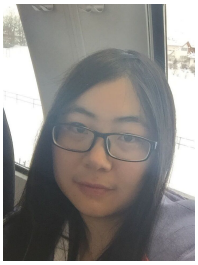
This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3188005

CoRR, vol. abs/2103.07189, 2021. arXiv: 2103.07189. [Online]. Available: https://arxiv.org/abs/2103.07189.

[44] A. Georges, L. Eeckhout, and D. Buytaert, "Java performance evaluation through rigorous replay compilation," in *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008, pp. 367–384.

[45] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," *SIGPLAN Not.*, vol. 48, no. 4, pp. 219–228, Mar. 2013, ISSN: 0362-1340. DOI: 10.1145/2499368.2451141.

Mostafa Jangali is a PhD student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He fast-tracked MSc. of Computer Science at Concordia University, and received his BSc. in Engineering of Information Technology at Sharif University of Technology, Tehran, Iran. Contact him at m_jangal@encs.concordia.ca.



Yiming Tang is a Postdoctoral Fellow in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Her research areas are software transformation, software testing and static analysis. Her research has been published in ICSE, SCAM, FASE, and SCP, among other prominent software engineering conferences and journals. She received her BEng. from Xidian University, China, her MSc. from the University of Manchester, UK and PhD in Computer Science from City University of New York, USA. Contact her at t_yiming@encs.concordia.ca.

Niclas Alexandersson is a master student of Software Engineering at Chalmers University of Technology, and a former intern at Opera and Google.

Philipp Leitner is an Associate Professor of Cloud-Based Software Engineering at Chalmers University of Technology in Gothenburg (Sweden), where he also leads the Internet Computing and Emerging Technologies Lab (ICET-lab). He holds a PhD degree from Vienna University of Technology. Philipp's research interest lie in empirical software engineering, with a particular focus on software performance engineering for Web-, service-, and cloud-based systems. Contact him at philipp.leitner@chalmers.se; http://philippleitner.net/.

Jinqiu Yang is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Her research interests include automated program repair, software testing, quality assurance of machine learning software, and mining software repositories. Her work has been published in flagship conferences and journals such as ICSE, FSE, EMSE. She serves regularly as a program committee member of international conferences in Software Engineering, such as ASE, ICSE, ICSME and SANER. She is a regular reviewer for Software Engineering journals such as EMSE, TSE, TOSEM and JSS. Dr. Yang obtained her BEng from Nanjing University, and MSc and PhD from University of Waterloo. More information at: https://jinqiuyang.github.io/.

Weiyi Shang is a Concordia University Research Chair at the Department of Computer Science. His research interests include AIOps, big bata software engineering, software log analytics and software performance engineering. He serves as a Steering committee member of the SPEC Research Group. He is ranked top worldwide SE research stars in a recent bibliometrics assessment of software engineering scholars. He is a recipient of various premium awards, including the SIGSOFT Distinguished paper award at ICSE 2013 and ICSE 2020, best paper award at WCRE 2011 and the Distinguished reviewer award for the Empirical Software Engineering journal. His research has been adopted by industrial collaborators (e.g., BlackBerry and Ericsson) to improve the quality and performance of their software systems that are used by millions of users worldwide. Contact him at shang@encs.concordia.ca; https://users.encs.concordia.ca/~shang/.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Concordia University Library. Downloaded on July 05,2022 at 23:40:04 UTC from IEEE Xplore. Restrictions apply.